MICROPOWER SERIES

GRAPHICS, SOUND & MUSIC

for the Commodore 64®

Don Cassel

- Covers all features of Commodore graphics, including sprite and keyboard graphics.
- Enhance and create your own arcade games.
- Create sound and music, from one voice sound to harmonies and music instruments.
- Learn to use Commodore's Super Expander cartridge to get everything you can from the 64's sound and graphics.
- Dozens of programs included.

GRAPHICS, SOUND & MUSIC for the Commodore 64® Don Cassel

wcb

GRAPHICS, SOUND & MUSIC

for the Commodore 64®

MICROPOWER SERIES

GRAPHICS, SOUND & MUSIC

for the Commodore 64®

Don Cassel
Humber College

wcb
Wm. C. Brown Publishers
Dubuque, Iowa

Consulting Editor:
Edouard J. Desautels
University of Wisconsin—Madison

MICROPOWER SERIES

Cover photo by Bob Coyle.

Figures 1.1 and 1.2 and Table 2.1 are provided courtesy of Commodore Business Machines.

Copyright © 1985 by Wm. C. Brown Publishers. All rights reserved

Library of Congress Catalog Card Number: 84–71557

Book
ISBN 0–697–00423–6

Book and Diskette
ISBN 0–697–00422–8

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher.

Printed in the United States of America
10 9 8 7 6 5 4 3 2 1

Contents

Introduction xi

PART ONE Graphics 1

1 Keyboard Graphics 3

GRAPHIC CHARACTERS ON THE KEYBOARD 3
- *Normal Mode* 4
- *Reverse Graphics* 4
- *Color Graphics* 5

HOW TO TYPE GRAPHIC PROGRAMS 5
PRINT STRINGS 5
SCREEN AND BORDER COLORS 6
CLEARING THE SCREEN 6
I'D RATHER BE SAILING 6
ALPHABETIC KEYBOARD GRAPHIC 8
TEST YOUR REACTIONS 10
CREATING BUSINESS GRAPHS: BAR CHARTS 12
- *Drawing the Chart* 13
- *The Data* 13

2 Using POKE to Create Graphics 17

SCREEN ADDRESSING 17
COLOR ADDRESSING 18
IMPROVED SCREEN AND COLOR ADDRESSING 20
DRAWING A HORIZONTAL LINE 21
DRAWING A VERTICAL LINE 21
DRAWING A BORDER 21
LUNAR LANDER: COMBINING TWO IMAGES ON THE SCREEN 23
USING PEEK AND POKE 25
USING PRINTS AND POKES IN A DIGITAL CLOCK 27

3 *Animation* 31

PRINCIPLES OF ANIMATION 31
ANIMATING WITH PRINT STATEMENTS 31
HORIZONTAL MOVEMENT: LEFT TO RIGHT 32
HORIZONTAL MOVEMENT: RIGHT TO LEFT 32
VERTICAL MOVEMENT: TOP TO BOTTOM 33
VERTICAL MOVEMENT: BOTTOM TO TOP 33
SATURN V ROCKET 34
ANIMATING MULTIPLE OBJECTS: THE TRAIN 35
ANIMATING WITH POKE STATEMENTS 38
HORIZONTAL MOVEMENT: POKING LEFT TO RIGHT 38
HORIZONTAL MOVEMENT: POKING RIGHT TO LEFT 39
VERTICAL MOVEMENT: POKING TOP TO BOTTOM 40
VERTICAL MOVEMENT: POKING BOTTOM TO TOP 40
STICKMAN: ANIMATING SEVERAL CHARACTERS SIMULTANEOUSLY 41
BOUNCING BALL 43

4 *Sprite Graphics* 47

WHAT IS A SPRITE? 47
SPRITE DESIGN 48
CONVERTING THE SPRITE TO NUMBERS 48
SPRITE DATA IN THE PROGRAM 50
DISPLAYING A SPRITE 51
SOME SAMPLE SPRITE DESIGNS 51
SOME HINTS ON CREATING SPRITES 62
SELECT SPRITE PROGRAM 62

5 *Sprite Graphics and Memory* 65

SPRITE REGISTERS 65
AVAILABLE MEMORY FOR SPRITES 66
STORING MULTIPLE SPRITES 66
ENABLING AND DISABLING SPRITES 67
ENABLING A SINGLE SPRITE 67
DISABLING SPRITES 68
ENABLING SEVERAL SPRITES 68
  Enabling Multiple Sprites 68
  Enabling Selective Sprites 68
DISABLING SELECTIVE SPRITES 69
UNDERSTANDING BOOLEAN AND'S AND OR'S 69
  OR to Enable a Sprite 69
  AND to Disable a Sprite 70
SPRITE MOVEMENT 70
SCREEN POSITIONING OF SPRITES 71
SPRITE ANIMATION 72
MOVING A SPRITE HORIZONTALLY 72
MOVING A SPRITE VERTICALLY 73
SPEEDING UP YOUR SPRITES 74
USING THE MSB REGISTER FOR FULL SCREEN WIDTH 75

SPRITE COLOR 76
SPRITE SIZE 77
SELECTING SIZE AND COLOR IN THE SPRITE PROGRAM 78

6 Advanced Sprite Graphics 85
SPRITE PRIORITIES 85
CARS PASSING ON THE HIGHWAY 85
BACKGROUND TO SPRITE PRIORITY 87
CARS ON HIGHWAY REWRITTEN 87
BACKGROUND PRIORITY REGISTER 90
SUNSET 90
SPRITE COLLISION DETECTION 92
CHOPPER COLLISION 93
SPRITE-TO-BACKGROUND COLLISION 94
COLLISIONS OFF THE SCREEN 95
MULTICOLORED SPRITES 95
  Setting Multicolored Mode 95
  Multicolor Registers 95
  Multicolored Sprite Shapes 96
CREATING A MULTICOLOR FLAG 96

PART TWO Sound and Music 99
7 The Basics of Music Generation 101
GENERATING SOUND 101
VOLUME SETTING 102
FREQUENCY SETTINGS 102
MUSIC OCTAVES 103
DATA STATEMENTS FOR NOTE VALUES 104
"SURPRISE SYMPHONY" (HAYDN) 105
CONTROLLING NOTE DURATION 106
SECOND "SURPRISE SYMPHONY" 107
ATTACK AND DECAY SETTINGS 107
SUSTAIN AND RELEASE SETTINGS 109
COMBINING ATTACK/DECAY AND SUSTAIN/RELEASE SETTINGS 109
"TAKE ME OUT TO THE BALL GAME" 110
WAVEFORM SETTINGS 111
  Triangle Waveform 111
  Sawtooth Waveform 112
  Pulse Waveform 112
  Noise Waveform 112
USING THE WAVEFORM SETTINGS 113
"LARGO" 114
"WILLIAM TELL OVERTURE" 114
"PIZZICATO POLKA" 115

8 More Music Techniques 117
USING DECIMAL FREQUENCY NUMBERS 117
RUNNING UP THE SCALE 118
CHANGING THE OCTAVE 119

REPEATING THE SCALE 119
STORING NOTES IN AN ARRAY 120
REVERSING THE SCALE 121
TWO- AND THREE-VOICE MUSIC 121
"AMAZING GRACE" 122
IMPROVING THE PROGRAM 123
  Playing in a Higher Octave 124
  Getting a Smoother Tone 124
  Adding the Drone 124
"AMAZING GRACE" WITH BAGPIPES 124
MULTIVOICE SOUND WITH INDEPENDENT DURATION 125
"SKATER'S WALTZ" 126

9 Advanced Music Techniques 131
FILTERS 131
  Low-Pass Filter 131
  High-Pass Filter 131
  Bandpass Filter 131
  Band Reject Effect 132
USING THE FILTERS 132
SELECTING THE VOICE TO FILTER 132
SETTING THE FILTER MODE 132
CHOOSING THE CUTOFF FREQUENCY 133
ADDING RESONANCE 134
"WILLIAM TELL" WITH FILTERING 134
SIMPLIFYING MUSIC DATA ENTRY 135
"THE ENTERTAINER" 137

10 Sound Effects 141
THE BASIC SOUND EFFECT TECHNIQUE 141
SOUND VARIATIONS 142
USING REVERSE LOOPS 143
USING TWO SOUND LOOPS 143
MARTIANS 144
USING ALTERNATING TONES 145
SIREN 145
GENERATING RANDOM SOUNDS 146
COMPUTER TALK 146
WHITE NOISE 147
STEAM ENGINE 147
USING THE RANDOM FUNCTION WITH WHITE NOISE 148
GUNFIRE 148
COMBINING SOUND EFFECTS 148
BOMB 149

11 The VIC and SID Show 151
COMBINING GRAPHICS AND SOUND 151
VIC AND SID 151

Appendixes 153

- **A: COMMODORE 64 COLOR CODES AND COMBINATIONS 153**
  - Border and Background Colors 153
  - Excellent Color Combinations 154
- **B: SPRITE REGISTER ADDRESSES 155**
  - Sprite Registers 155
  - Memory Allocation for Sprite Data 155
  - Sprite Enable/Disable Register V+21 155
  - Sprite X and Y Coordinate Registers 156
  - Most Significant Bit Register V + 16 156
  - Sprite Color Registers and Codes 156
  - Color Codes 156
  - Registers for Sprite Expansion 157
  - Background Priority Register V + 27 157
  - Sprite-to-Sprite Collision Register V + 30 157
  - Sprite-to-Background Collision Register V + 31 157
  - Sprite Multicolor Register V + 28 158
- **C: SOUND REGISTERS 159**
  - POKE Addresses for SID Sound Settings 159
  - POKE Values for Octave Four Frequencies 159
  - Values for Attack/Decay Settings 160
  - Values for Sustain/Release Settings 160
  - Values for Waveform Settings 161
  - POKE Values for Pulse Rate 161
  - Voice-to-Filter S + 23 161
  - Set Filter Type S + 24 162
  - Cutoff Frequency 162
  - High Order Bits S + 22 162
  - Resonance and Voice-to-Filter S + 23 162
- **D: VOICE FREQUENCY POKE VALUES 163**
  - Voice Frequency POKE Values 163

Index 165

Introduction

Welcome to the exciting world of graphics and sound for the Commodore 64 personal computer. If you have done some programming on the Commodore 64, then you will find this book to be a good source of information for creating effective programs that use both graphics and sound. If you are new to programming, this book can also introduce you to the capabilities of the Commodore 64. However, you may also want to read an introductory book on BASIC programming for the Commodore 64.

In this book, we will begin by looking at the graphics available to you directly from the keyboard. This is the easiest and most direct way of creating graphics. Next we will create graphics with POKE statements and then move on to animation. These techniques originated with the popular Commodore PET and are still useful methods for the creation of interesting graphics.

The next section discusses the use of sprite graphics. Sprites represent a quantum leap forward in the use of graphics on a personal computer. Although sprites can be more complex than keyboard graphics, they offer incredible flexibility and provide for animation, higher resolution, color, and three dimensions that are impossible with keyboard graphics.

The last section of the book covers the variety of sound and music capabilities built into the Commodore 64, and the terrific effects these features can produce.

Throughout the book are numerous programs written to demonstrate the features described. As each characteristic of the Commodore 64 is discussed, that feature is used in a program to demonstrate its use. These programs are included on a diskette if you purchased the package containing the software. I hope you find this book useful in your quest for writing more effective BASIC programs on the Commodore 64. If you have half as much fun trying the features of the computer as I did in writing the book, I'm sure you will be pleasantly surprised by the extensive capabilities designed into your personal computer.

PART ONE

Graphics

1

Keyboard Graphics

If you are new to creating graphics on the Commodore 64, then this chapter presents a friendly introduction. Keyboard graphics are the easiest to use although more difficult to demonstrate in a book. One of the nice features of keyboard graphics is that you can see immediately from the keyboard the shape of each graphic character. What you see on the key is what you get on the screen. Of course there are exceptions, but this is a good general rule.

GRAPHIC CHARACTERS ON THE KEYBOARD

The Commodore 64 keyboard (fig. 1.1) shows most of the graphic characters (fig. 1.2) that are available on the front face of the keys. Each key contains two graphic symbols. To select the symbol on the right of a key, press the shift key and hold it while you also press the graphic key. If you hold the shift while the "Q" key is pressed, you will get a solid blue circle on the screen.

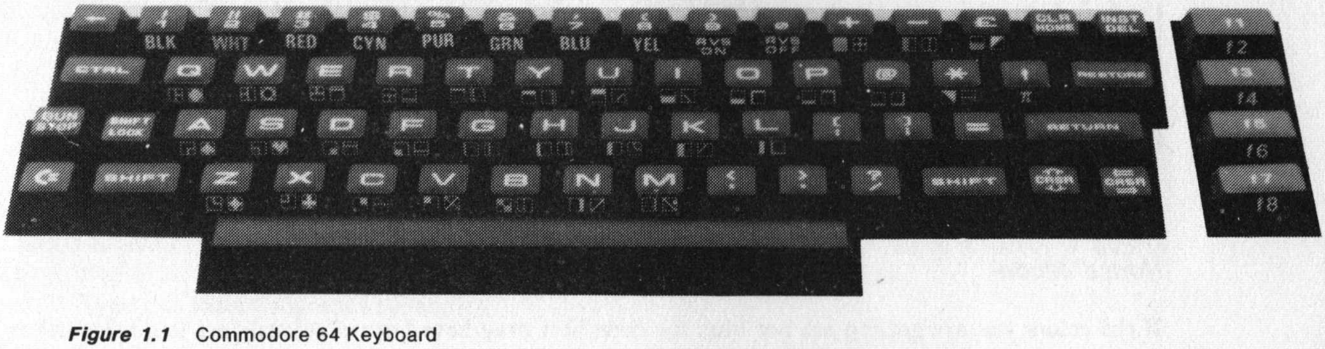

BLK
WHT
RED
CYN
PUR
GRN
BLU
YEL
RVS ON
RVS OFF
CLR HOME
INST DEL
CTRL
Q
W
E
R
T
Y
U
I
O
P
RESTORE
RUN STOP
SHIFT LOCK
A
S
D
F
G
H
J
K
L
RETURN
SHIFT
Z
X
C
V
B
N
M
SHIFT
CRSR
CRSR
f1
f2
f3
f4
f5
f6
f7
f8

Figure 1.1 Commodore 64 Keyboard

Graphic symbols on the left of the key can be selected by pressing and holding the Commodore key, then pressing the key containing the graphic character. The Commodore key is the one with the Commodore logo located beside the left shift key. Try holding the Commodore key while you press the * key. This time you will get a blue triangle on the screen.

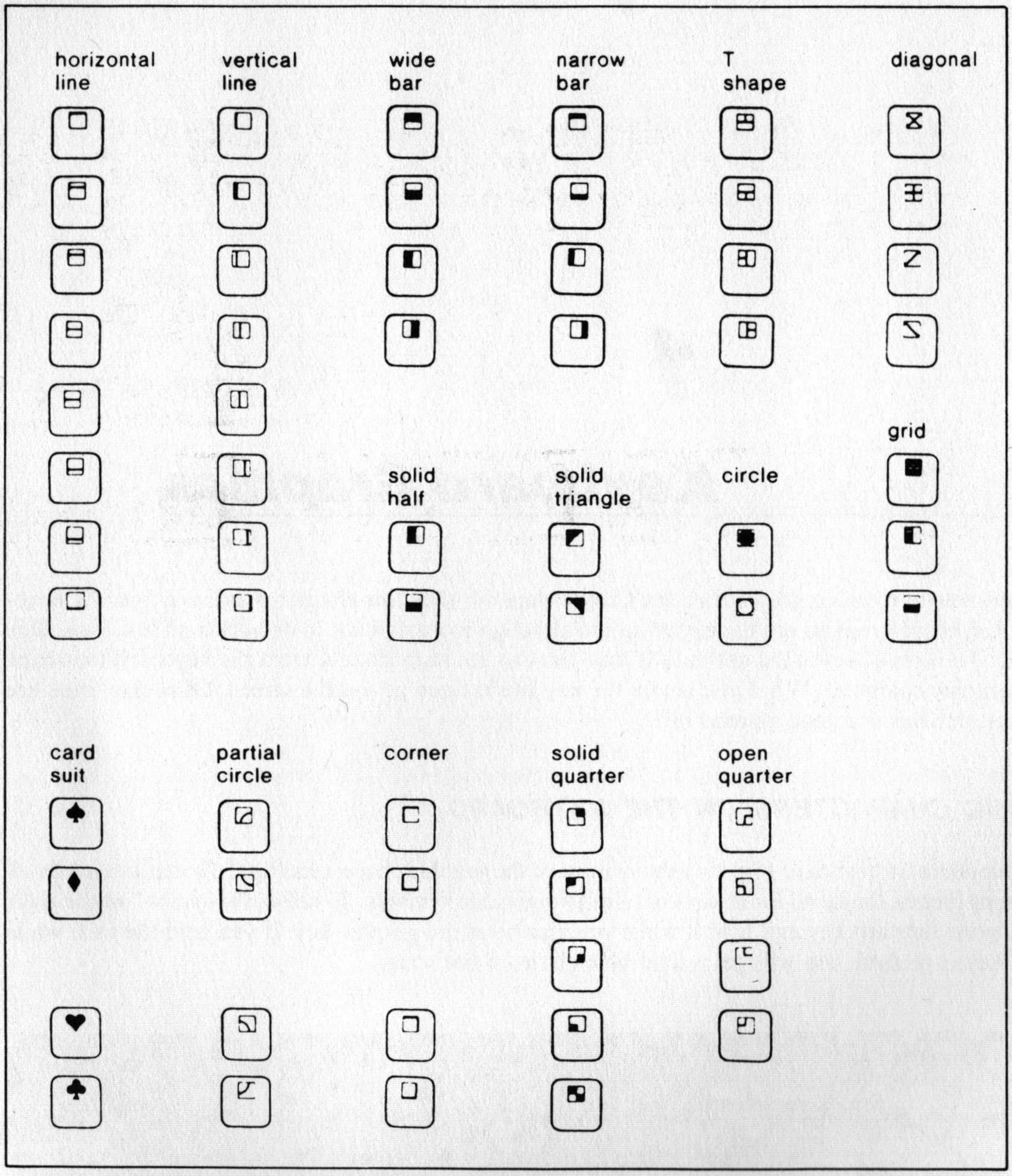

horizontal line
vertical line
wide bar
narrow bar
T shape
diagonal
grid
solid half
solid triangle
circle
card suit
partial circle
corner
solid quarter
open quarter

Figure 1.2 Keyboard Graphic Characters by Type

Normal Mode

If the colors you are getting are not blue the computer may have been previously set to display other colors. It can easily be returned to normal by holding down the RUN/STOP key and pressing RESTORE.

You also could have turned the computer off and then on again. At this point it doesn't really matter, but later RUN/STOP and RESTORE will be preferred, because turning the computer off will result in the loss of the program.

Reverse Graphics

At the left of the keyboard is the CTRL (control) key. When this key is held down, other features may be activated. Try this: Hold down CTRL and then press the 9 key. Now type some graphic characters as explained earlier. Notice that they are now displayed dark blue on a light blue background. This is called a reverse character.

The RVS ON (reverse on) letters on the 9 key are explained by this exercise. Pressing CTRL and RVS ON causes all future characters typed to be displayed in reverse mode. Try pressing the space bar. Even the space is reversed; it is now a solid character instead of an empty space.

Reversed characters are often used with nonreversed characters to create more effective images. To turn the reverse feature off, press CTRL and RVS OFF (on the zero key). Now any characters typed will be displayed normally.

Color Graphics

Any graphic character you use may also be displayed in a selected color. Press and hold the CTRL key and then press the WHT (white) key. It's on the 2 key. The cursor color will immediately turn white. Now type some graphics. They will also be white.

Press CTRL and GRN (green). All characters are now green. You may find that it is necessary to adjust your TV or monitor color to get a more natural result. In fact, some of the colors will not look all that great against a dark blue background, so later we will see how to change the background color to give more pleasing results.

HOW TO TYPE GRAPHIC PROGRAMS

Many of the programs in this book will contain a mixture of graphics, color, reverse, and cursor control characters. Since matrix printouts are not always easily interpreted, many of the programs will be listed with the following codes to ensure easy entry into your computer. To ensure some consistency in character representation the system used by *Compute!* magazine is adopted here.

Braces	Any characters in braces mean to press that key as indicated. {HOME} means to press the home key, {5 DOWN} means to press the cursor down key five times. {10 SPACES} means to press the space bar ten times and {RED} means to press CTRL and 3 to get the red color.
Underline	A character with the underline symbol such as <u>Z</u> means to press shift when that character is typed. A shifted Z would give the graphic diamond symbol.
Brackets	A character enclosed in brackets, such as [<Z>] means to press the Commodore key while the character is typed. This character would give the lower left corner graphic symbol.

PRINT STRINGS

Although creating graphics directly on the screen can be entertaining for a while, it is not very useful. But if these characters are included in print statement strings, they can be put to all sorts of uses in your program.

Type this short program and run it.

```
10 FOR I=1 TO 40          {YEL} means to press          looks like this
                                CTRL YEL
20 PRINT "{YEL} ●O";
                                                           " π ●O"
30 NEXT I
```

If all went well, you will get two rows of alternating solid and open yellow circles. Notice that the CTRL YEL displays a special symbol when it is used in a string, instead of changing the cursor to yellow as it did without using strings. When the program stops, the cursor will still be yellow, so press the RUN/STOP and RESTORE before you continue.

Now list the program to see if it is still in memory. Try a few other colors in statement 20.

Table 1.1 Border and Background Colors

Border color		**POKE 53280,X**	
Background color		**POKE 53281,Y**	
		Values for X and Y	
Black	0	Orange	8
White	1	Brown	9
Red	2	Light Red	10
Cyan	3	Dark Gray	11
Purple	4	Medium Gray	12
Green	5	Light Green	13
Blue	6	Light Blue	14
Yellow	7	Light Gray	15

SCREEN AND BORDER COLORS

Now try another revision to the above program. Type in the following program and run it. If a program still exists in the computer, press RUN/STOP and RESTORE and then enter NEW to clear memory.

```
10 POKE 53280,6:POKE 53281,1
20 FOR I=1 TO 40          [CTRL RED]          [CTRL GRN] looks like this

30 PRINT "{RED} ● {GRN} ●";
                                                        "[£]●[↑]●"
40 NEXT I
```

Now you will have a blue border and a white screen with red and green circles on two lines. The POKEs in statement 10 are setting the border and background (screen) colors to blue and white, respectively. Table 1.1 gives a complete list of poke codes that may be used to set the display to the desired colors.

CLEARING THE SCREEN

Now add the following statement to the above program.

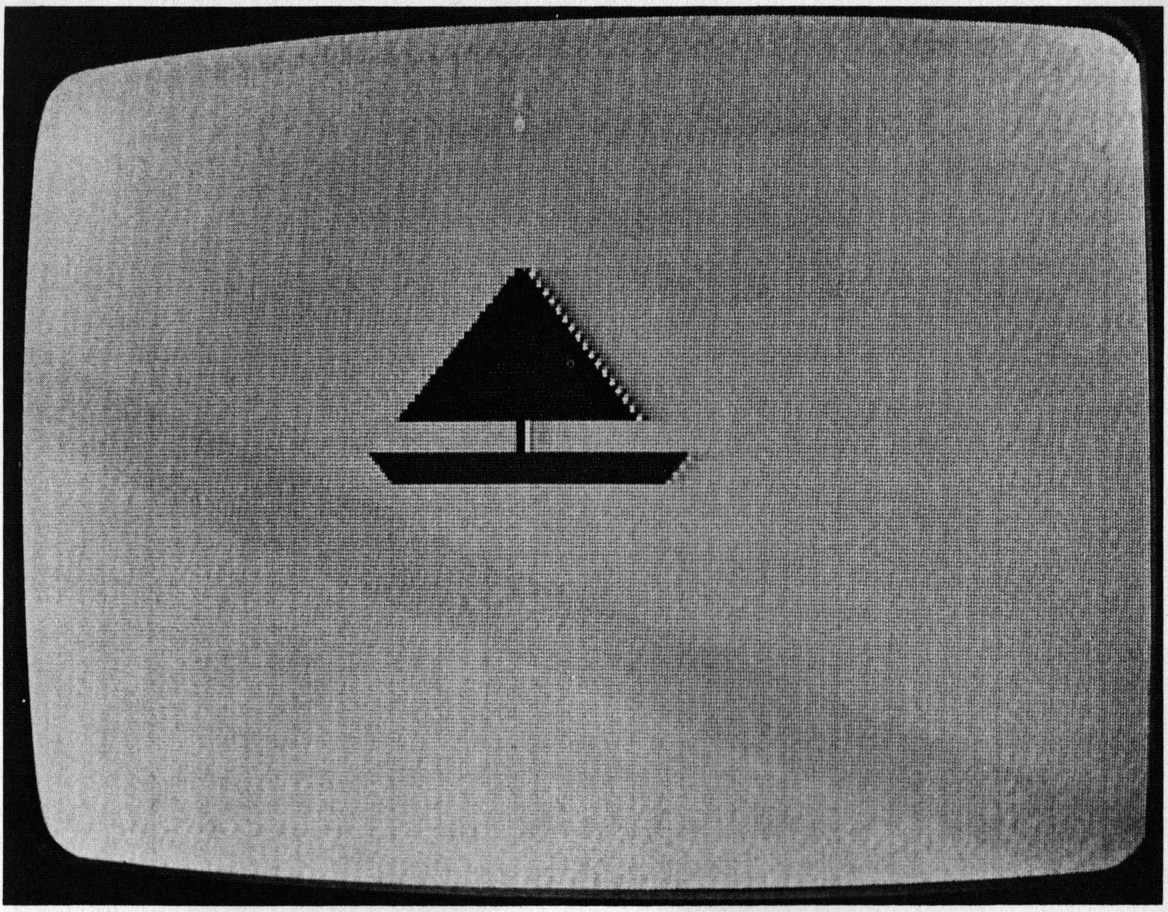
```
5 PRINT CHR$(147);
```

Run the program. This time the screen will be cleared before the circles are displayed. It's a good idea to use a statement like this at the beginning of graphic programs, so that your output will always be on a clear screen.

I'D RATHER BE SAILING

Now let's put together the graphic and color characters to create a sailboat. The program that follows displays a graphic of a blue boat with a red sail against an azure blue (it's really cyan) sky. The sky color is determined by POKEing both the border and background colors with a 3.

The key to entering this program successfully is to make sure all of the spaces, color control characters, and reverse characters are each entered as shown. It may help to refer back to the page explaining the method for typing these programs.



```
10 REM SAILBOAT
20 PRINT CHR$(147):PRINT:PRINT:PRINT:PRINT:PRINT:PRINT
30 POKE 53280,3:POKE 53281,3
40 I=13
50 PRINT TAB(I);"{5 SPACES}{RED}{RVS}£ [<*>]"
60 PRINT TAB(I);"{4 SPACES}{RED}{RVS}£{2 SPACES}[<*>]"
70 PRINT TAB(I);"{3 SPACES}{RED}{RVS}£ {4 SPACES}[<*>]"
80 PRINT TAB(I);"{2 SPACES}{RED}{RVS}£ {6 SPACES}[<*>]"
90 PRINT TAB(I);"{1 SPACE}{RED}{RVS}£ {8 SPACES}[<*>]"
100 PRINT TAB(I);"{6 SPACES}{RED}[<J>]"
110 PRINT TAB(I);"{BLU}*{RVS}{11 SPACES}{OFF} £"
120 PRINT:PRINT:PRINT
```

Comments on the Program

Lines	
20	Clears the screen and moves the cursor down 6 lines.
30	Sets the border and background to cyan.
40	Sets I to 13 for a tab value.
50–90	Creates a red sail.
100	This is the bottom part of the mast.
110	This is the boat itself.
120	Moves the cursor 3 lines below the graphic.

Challenge

When you get tired of looking at this work of art, press RUN/STOP and RESTORE. Now list the program and try a challenge. How about adding a flag to the boat? Or a tiller? Use your imagination and see what interesting effects you can create.

Table 1.2 Excellent Color Combinations

		CHARACTER COLOR															
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	0		X		X	X			X	X		X	X	X	X	X	X
	1	X		X		X	X	X			X		X	X		X	X
S	2		X						X	X		X					
C	3	X						X									
R	4	X															
E	5	X													X		
E	6		X		X											X	X
N	7	X		X							X		X	X			
	8		X	X					X		X						
C	9		X						X	X		X					X
O	10			X							X						
L	11	X	X						X					X	X		X
O	12	X	X										X				X
R	13	X					X						X				
	14	X	X		X			X									
	15	X	X	X				X					X	X			

0	Black
1	White
2	Red
3	Cyan
4	Purple
5	Green
6	Blue
7	Yellow
8	Orange
9	Brown
10	Light Red
11	Gray 1
12	Gray 2
13	Light Green
14	Light Blue
15	Gray 3

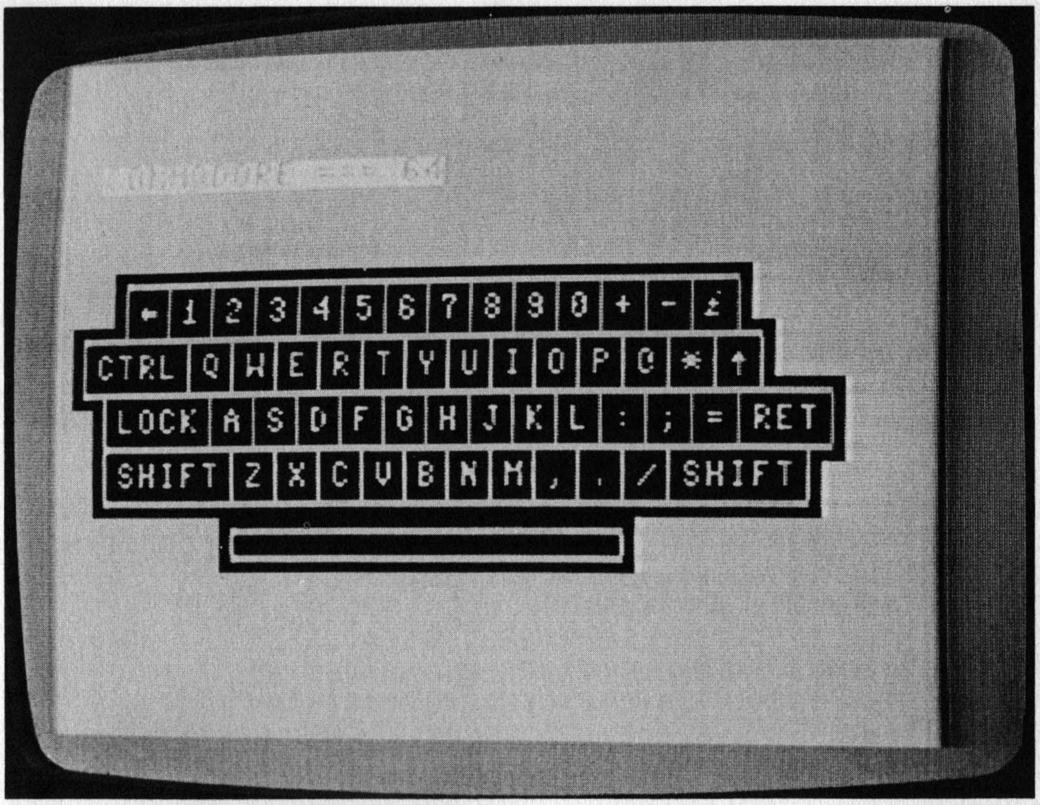
ALPHABETIC KEYBOARD GRAPHIC

The next program uses keyboard graphics to produce a graphic of itself. Since the screen size is limited to 40 characters on a line, some of the control keys, such as the cursor controls, are not included. Otherwise the graphic is a reasonably good image of the Commodore 64's keyboard.

The program listing below shows how the program was written. Notice the use of the reverse character in lines 40 to 220 to create black keys with white images. The black color is selected in line 110 immediately after the reverse character. Once set to a color, all of the following characters that are printed by the program will also be black.

The colors selected for this graphic are close to the actual color of the Commodore 64. The color combination works out quite well. However, not all combinations of colors will look good on your display screen. This will partly depend on the TV or monitor you are using, but some color combinations just don't work well together. Refer to table 1.2 for a chart of excellent combinations.

Expressions like {13{C[<R>]}} in this program mean to type 13 groups of the shifted C and the Commodore key with the letter R.


COMMODORE === 64
← 1 2 3 4 5 6 7 8 9 0 + - £
CTRL Q W E R T Y U I O P @ * ↑
LOCK A S D F G H J K L : ; = RET
SHIFT Z X C V B N M , . / SHIFT

```
10 REM ALPHABETIC KEYBOARD
20 POKE 53280,5:POKE 53281,15
30 PRINT "{CLR}{3 DOWN}"
40 PRINT "{2 SPACES}{RVS}{WHT}COMMODORE === 64"
50 PRINT:PRINT:PRINT
110 PRINT "{3 SPACES}{RVS}{BLK}[<A>]{13{C[<R>]}}C[<S>]"
120 PRINT "{3 SPACES}{RVS}B-B1B2B3B4B5B6B7B8B9B0B+B-BLB"
140 PRINT "{SPACE}{RVS}[<A>]C[<E>]C{13{[<E>][<R>]}}
    [<E>][<S>]"
150 PRINT "{SPACE}{RVS}BCTRLBBQBWBEBRBTBYBUBIBOBPB@B*B^B"
160 PRINT "{SPACE}{RVS}[<Z>][<R>]CCC{13{[<E>][<R>]}}
    [<E>]CC[<S>]"
170 PRINT "{2 SPACES}{RVS}BLOCKBABSBDBFBGBHBJBKBLB:B;B=
    BRETB"
180 PRINT "{2 SPACES}{RVS}[<Q>]CCCC{11{[<E>][<R>]}}{2{
    [<E>]C}}C[<R>][<X>]"
190 PRINT "{2 SPACES}{RVS}BSHIFTBZBXBCBVBBBNBMB,B.B/BSHIFTB"
200 PRINT "{2 SPACES}{RVS}[<Z>]CCCCC{11{[<E>]C}}CCCC[<X>]"
210 PRINT:PRINT "{8 SPACES}{RVS}[<A>]{17{C[<S>]}}"
220 PRINT "{8 SPACES}{RVS}[<Z>]{17{C[<X>]"
230 PRINT:PRINT
```

Comments on the Program

Lines	
20	Sets the border to green and the background to gray 3.
30	Clears the screen and moves the cursor down 3 lines.
40	Prints the Commodore logo in reverse white characters.
50	Leaves 3 blank lines.
60–220	Print the keyboard graphic in reverse black characters.

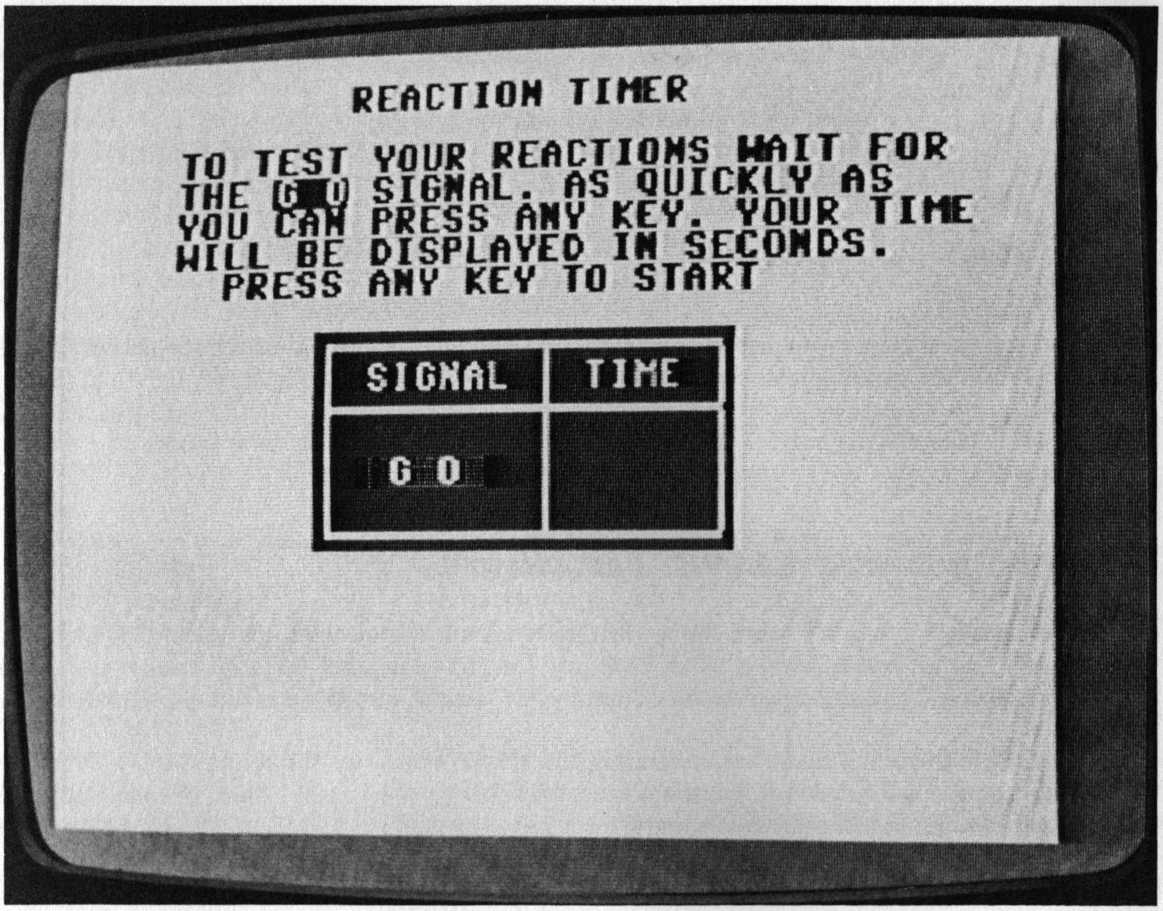
Challenge

Using cursor controls in a print string, display a character on the keyboard as if it were being pressed on the computer. Better still, display the character when the key is typed. This graphic could be developed into a program to teach typing or, at least, to demonstrate the various keys on the Commodore 64 keyboard.

TEST YOUR REACTIONS

The reaction timer is a program that tests your reflexes. You are prompted with a ready signal which is displayed within a scoreboard made up of reversed graphic characters. The ready signal gives you time to get on the alert. Then the GO is flashed on the scoreboard and the internal timer is started. As soon as a key is pressed, the time is taken and displayed on the scoreboard. What is your best time?

```
100 REM REACTION TIMER
110 GOSUB 800
150 PRINT TAB(11)"{BLU}{RVS}[<A>]CCCCCCCC[<R>]CCCCCC[<S>]"
160 PRINT TAB(11)"{RVS}B SIGNAL B TIME B"
170 PRINT TAB(11)"{RVS}[<Q>]CCCCCCCC+CCCCCC[<W>]"
180 PRINT TAB(11)"{RVS}B{8 SPACES}B{6 SPACES}B"
182 PRINT TAB(11)"{RVS}B{8 SPACES}B{6 SPACES}B"
184 PRINT TAB(11)"{RVS}B{8 SPACES}B{6 SPACES}B"
190 PRINT TAB(11)"{RVS}[<Z>]CCCCCCCC[<E>]CCCCCC[<X>]"
195 GOSUB 700
200 GOSUB 500
210 GOSUB 540
220 INPUT "{4 DOWNS}{BLU}TRY AGAIN (Y/N)"; A$
230 IF LEFT$(A$,1)="Y" THEN 100
240 STOP
500 REM DISPLAY READY SIGNAL
510 GOSUB 600
520 PRINT TAB(13)"{RED}{RVS}READY";
530 GOSUB 700
535 RETURN
540 REM DISPLAY GO SIGNAL
542 GOSUB 600
544 PRINT TAB(13)"{GRN}{RVS} G O {OFF}";
546 OT=TI
548 GET A$: IF A$="" THEN 548
550 NT=TI
551 ET=INT((NT-OT)/60*1000)/1000
552 PRINT SPC(3); ET
554 RETURN
600 PRINT "{HOME}{13 DOWNS};
610 RETURN
700 REM TIME DELAY
710 FOR I=1 TO 1500:NEXT
720 RETURN
800 REM INSTRUCTIONS
810 POKE 53280,14:POKE 53281,1
820 PRINT CHR$(147)
830 PRINT TAB(12);"{BLU}REACTION TIMER}
840 PRINT
850 PRINT TAB(5);"TO TEST YOUR REACTIONS WAIT FOR"
860 PRINT TAB(5);"THE {RVS}G O{OFF} SIGNAL. AS QUICKLY AS"
```



```
        REACTION TIMER

TO TEST YOUR REACTIONS WAIT FOR
THE (GO) SIGNAL. AS QUICKLY AS
YOU CAN PRESS ANY KEY. YOUR TIME
WILL BE DISPLAYED IN SECONDS.
   PRESS ANY KEY TO START

       | SIGNAL | TIME |
       |   GO   |      |
```

```
870 PRINT TAB(5);"YOU CAN PRESS ANY KEY. YOUR TIME"
880 PRINT TAB(5);"WILL BE DISPLAYED IN SECONDS."
890 PRINT TAB(5);"PRESS ANY KEY TO START"
900 PRINT
910 GET A$:IF A$="" THEN 910
920 RETURN
```

Comments on the Program

Lines	
110	Goes to the instruction subroutine.
150–190	Display the reaction timer scoreboard.
195	Goes to the time delay subroutine.
200	Goes to READY signal subroutine.
210	Goes to GO signal subroutine.
220–240	Determine if player wants to try again.
500–535	Display READY signal subroutine.
510	Goes to position cursor in scoreboard.
520	Prints READY signal.
530	Time delay.
540–554	GO signal subroutine.
542	Goes to position cursor in scoreboard.
544	Prints GO signal.
546	Stores time (TI) in old time (OT).
548	Gets player's response to GO signal.
550	Stores time (TI) in new time (NT).
551	Calculates elapsed time (ET) in seconds.

552	Prints elapsed time in scoreboard.
600–610	Positions cursor in scoreboard subroutine.
700–720	Time delay subroutine.
800–920	Instruction subroutine.
810	Blue border and white background.
820	Clears screen.
830–890	Print instructions.
910	Waits for player to read instructions.

Challenge

Why not revise this program to display READY, SET, and GO? You could use the colors red, yellow, and green, just like a traffic light, for this. Better still, create a graphic of a traffic light and turn on each color for the countdown.

Another possibility is to store and display the best time and challenge your friends to a reaction time match.

CREATING BUSINESS GRAPHS: BAR CHARTS

Bar charts are useful tools for representing changes over time in a business operation. Creating a bar chart can give an immediate view of sales, profit and loss, travel expense patterns, and so on. For this application, we will create a program that displays the value of a stock in monthly intervals for the period of a year.

One of the important first steps in creating any graph is to design the screen display. What you need is a picture of the output you want to create. With this picture you will, then, be more successful in developing the program to display, in graph form, the value of the stock. Figure 1.3 shows the initial design of the screen as we want it to appear from the program.

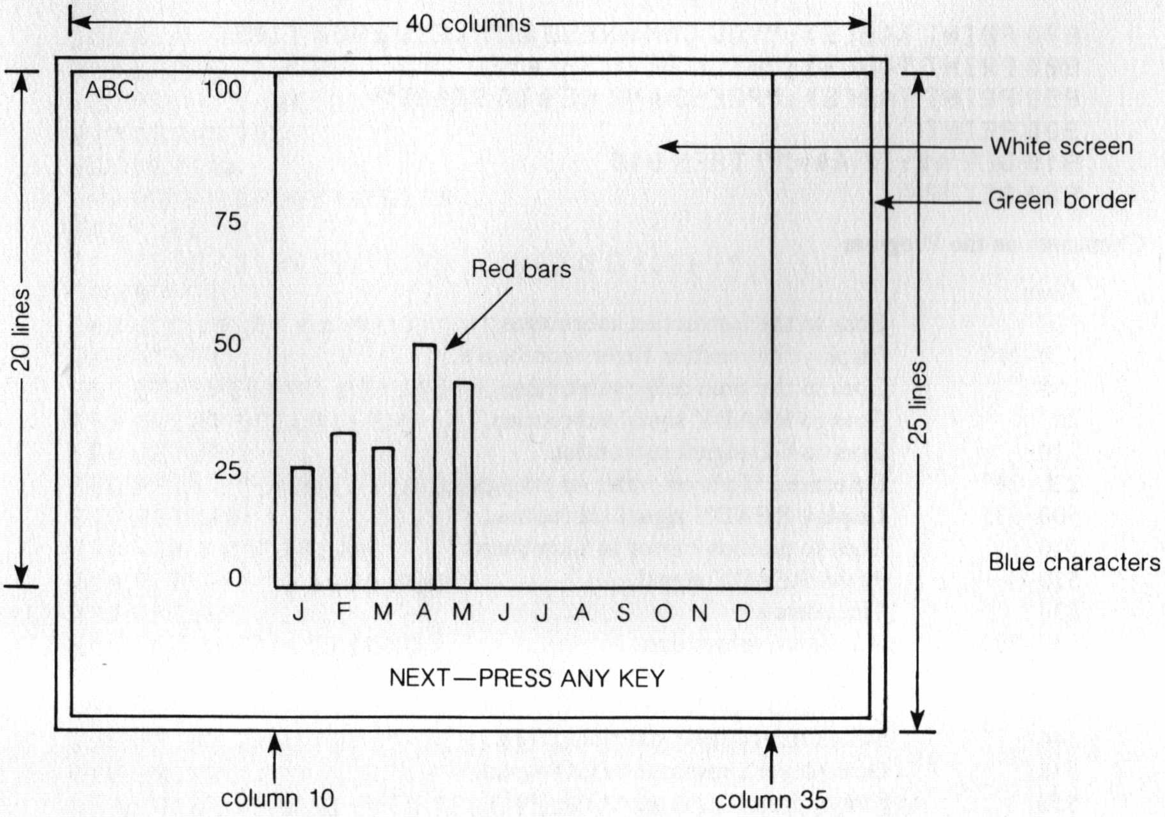


Figure 1.3 Stocks Program Screen Design

When designing a screen display such as this one, it is necessary to plan in detail how the entire screen space will be managed. With the Commodore 64 we can work with 40 columns and 25 lines when keyboard graphics are used. The bar graph itself will occupy a space of 20 lines and 24 columns, 2 columns for each month represented. One column for each month will be used for the bar, and the other will provide a space between the months.

The remaining 5 lines below the graph will display a legend labeling each month by letter, and also a prompt to indicate how to proceed to the next screen of data.

The 9 columns to the left of the chart are used to display the name of the stock in the upper left corner (home position) and a scale along the vertical axis which represents the stock's range of values. In this example, the range is from 0 to 100. This range will be specified in the data and will be plotted relative to the space available on the screen.

Drawing the Chart

The lines for the chart's outline are drawn using the following keys:

	Graphic Character	*Type*
Left column	Vertical bar on M	[<M>]
Right column	Vertical bar on G	[<G>]
Bottom line	Horizontal bar on T	[<T>]

After the outline is drawn, the letters for each month are displayed and so is the prompt at the bottom of the screen.

Next the scale at the left is printed. To do this, it is necessary to read the maximum value for the scale from the data. This value is then adjusted to create a scaling factor, so that the value can be equally scaled along the left vertical axis. The stock's name is also printed at this time.

Finally the monthly values of the stock are displayed in bar chart form. The red bars are formed by using the reverse space character.

The Data

Data for each stock is represented in the following form

```
name,scale,value1, . . . , value12
```

For example, stock ABC might have the following values:

```
1000 DATA ABC,100,55,57,60,58,54,50,35,32,45,52,67,71
```

The scale value of 100 means the stock is to be graphed between 0 and 100 dollars. In some cases, such as early in the year, there will not be values for later months; therefore, a −1 is placed after the value for the last month.

```
1010 DATA DEF,50,25,30,35,40,-1
```

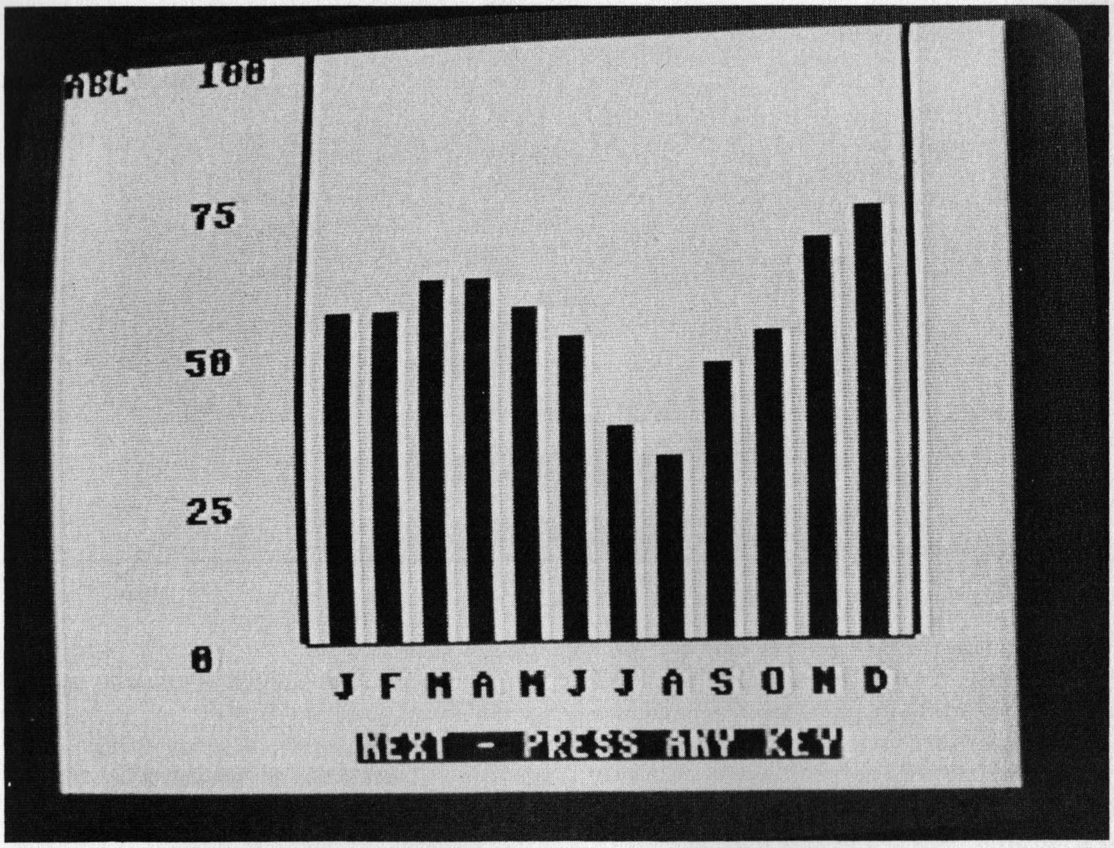
When this data is read, only the first five months will display. Then as each month progresses, you insert the new data in the data statement before the −1. When December is reached, the −1 should be deleted.

```
100 REM STOCK CHART
110 D$="{25 DOWNS}"
120 READ N$
130 IF N$="END" THEN PRINT "{HOME}"; D$:STOP
140 GOSUB 500
150 READ SC
```

```
160 GOSUB 700
170 GOSUB 800
180 GET A$:IF A$="" THEN 180
190 GOTO 120
500 REM SCREEN PAGE
510 POKE 53280,5:POKE 53281,1
520 PRINT CHR$(147);
530 FOR I=1 TO 20
540 PRINT TAB(10); "{BLU}[<M>]"; TAB(36); "[<G>]"
550 NEXT I
560 PRINT "{HOME}"; LEFT$(D$,20);
570 FOR I=11 TO 35
580 PRINT TAB(I); "[<T>]";
590 NEXT I
600 PRINT
610 PRINT TAB(12);"J F M A M J J A S O N D"
620 PRINT
630 PRINT TAB(13);"{RVS}NEXT - PRESS ANY KEY"
640 RETURN
700 REM PRINT SCALE
710 PRINT "{HOME}";TAB(5);SC
720 PRINT LEFT$(D$,4);TAB(5);INT((SC/4)*3+.5)
730 PRINT LEFT$(D$,4);TAB(5);INT((SC/2)+.5)
740 PRINT LEFT$(D$,4);TAB(5);INT((SC/4)+.5)
750 PRINT LEFT$(D$,4);TAB(5);0
760 PRINT "{HOME}";N$
770 RETURN
800 REM DISPLAY BAR CHART
810 FOR M=1 TO 12
820 READ A
830 IF A=-1 THEN 890
840 FC=21-INT(20/(SC/A)+.5)
850 FOR K=20 TO FC STEP -1
860 PRINT "{HOME}";LEFT$(D$,K-1);T AB(10+M*2);
    "{RVS}{RED}{SPACE}"
870 NEXT K
880 NEXT M
890 RETURN
1000 DATA ABC,100,55,57,60,58,54,50,35,32,45,52,67,71
1010 DATA DEF,50,25,30,35,40,-1
1020 DATA XYZ,10,3,3.50,4,3,2.5,2,4,6,-1
1030 DATA END
```

Comments on the Program

Lines	
100	Sets up 25 down characters for later use.
120	Reads name of stock.
130	If last stock, positions the cursor to the end of the screen before stopping the program.
140	Subroutine 500, sets up the screen image.
150	Reads the scaling factor.
160	Subroutine 700, displays the vertical scale.
170	Subroutine 800, displays the bar chart.
180–190	Wait loop until a key is pressed to display the next chart on the screen.
500–640	Subroutine to set up the screen graphic.
510	Sets border to green and background to white.



520–550	Clear the screen and display the two blue vertical lines that define the left and right margins of the chart.
560–590	Draw the horizontal line at the bottom of the chart.
600–610	Print the letters representing the months under the bottom line.
630	Prints the prompt message.
700–770	Subroutine to print the scale representing the dollar value of the stock.
710	Prints the maximum scale value at the top of the chart.
720–740	Print the scale values representing 3/4, 1/2, and 1/4 of the maximum.
750	Prints the zero value.
800–890	Subroutine to display the bars for each month.
810	Loop M represents each month.
820	Reads a single value for the month.
830	Checks for last value.
840	Scales stock's value to a range of 20.
850–870	Display bar for this month using reverse red spaces.
880	End of month loop.
1000–1030	Sample data. Scales of 100, 50, and 10 are used for stocks ABC, DEF, and XYZ, respectively.

Challenge

This program obviously has a lot of potential. As written it will only display data in monthly intervals. What if you wanted to use weekly or even daily intervals? One solution would be to simply change the line below the chart to represent the time interval desired.

Of course, if you used a different time interval, you might also want to display more than 12 bars. There is some additional space to do this by increasing the width of the screen. Another solution might be to eliminate the spaces between the bars. This would permit twice as much data to show.

Another possibility would be to use a disk or tape file to store the data. Then you could also provide an update to enter new values as they become available.

2

Using POKE to Create Graphics

Graphics that are produced with print strings can also be produced with POKE statements. The advantage of using POKEs is faster and more direct placement of the graphic on the screen. There are two reasons for a POKE operating faster: First, the command is itself inherently faster; and, secondly, it can directly access a screen address. A print must use cursor controls and/or multiple prints to get to the desired location on the screen.

However, POKE is not without it's faults. To use a poke the exact screen and color addresses must be known in order to position the character exactly where you want it on the screen. Whereas a print can just push the cursor around until you get where you want to be. A second nuisance is that the graphic character, or any other character for that matter, is represented by a special numeric POKE code and not the graphic characters themselves that you see on the keyboard.

With all its faults however, the POKE is still a very useful command for screen graphics and is especially good for animating small objects.

SCREEN ADDRESSING

As shown in figure 2.1, the Commodore 64 screen memory consists of 1,000 positions for displaying characters. These positions are memory addresses 1024 to 2023. A number, letter, or graphic character may be directly placed anywhere on the screen, by poking the screen address with the character's poke value. Poke values are given in table 2.1.

For example, to place a heart symbol at the top-left corner of the screen use the statement

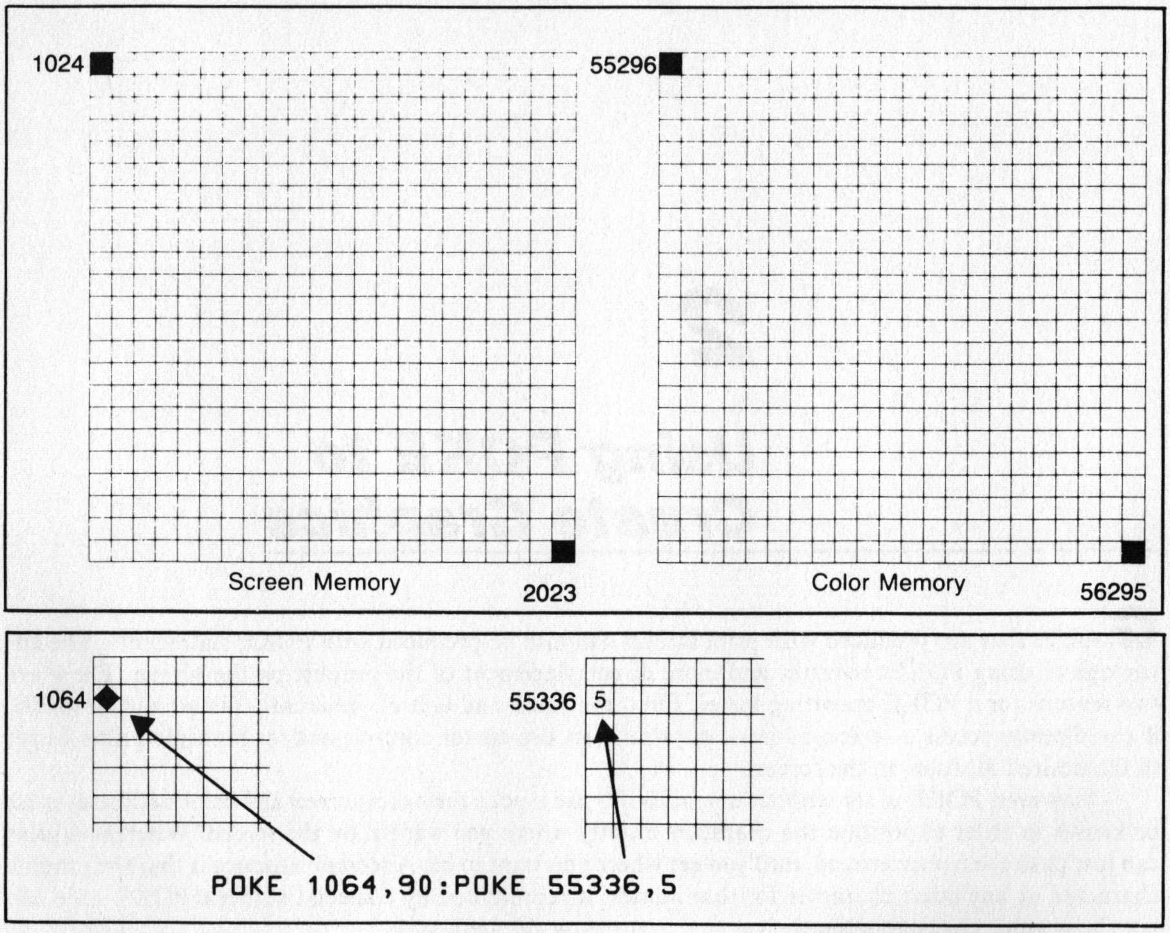
```
POKE 1024,83
```

The statement

```
POKE 2023,122
```

would place a corner symbol at the bottom right corner of the screen.

If you tried these pokes, the symbols may not have appeared on the screen as I suggested. Don't panic! There is one more part of the puzzle to be solved. So read on about color memory addresses.



```
POKE 1064,90:POKE 55336,5
```

Black	0	Orange	8
White	1	Brown	9
Red	2	Light Red	10
Cyan	3	Dark Gray	11
Purple	4	Medium Gray	12
Green	5	Light Green	13
Blue	6	Light Blue	14
Yellow	7	Light Gray	15

Figure 2.1 Commodore 64 Display Screen Addressing

COLOR ADDRESSING

The reason the pokes just described may not have placed graphics on the screen is that the Commodore 64 also needs to know what color the character is supposed to be. To determine color, the computer looks at its color addresses. These color addresses look like the screen memory except that they will contain only color codes from 0 to 15.

The color memory is located between addresses 55296 and 56295. As figure 2.1 shows, the color memory is arranged just like the screen memory. The first color address (55296) supplies the color for the first screen address (1024), the second color address (55297) gives the color code for the second screen location (1025), and so on, and so on, and so on.

Now let's try those pokes again, but this time we will create a white heart and an orange corner symbol.

```
POKE 1024,83: POKE 55296,1
POKE 2023,122: POKE 55295,8
```

Table 2.1 Graphic Character Codes

SET 1	SET 2	POKE	SET 1	SET 2	POKE	SET 1	SET 2	POKE
@		0	R	r	18	$		36
A	a	1	S	s	19	%		37
B	b	2	T	t	20	&		38
C	c	3	U	u	21	'		39
D	d	4	V	v	22	(		40
E	e	5	W	w	23	)		41
F	f	6	X	x	24	*		42
G	g	7	Y	y	25	+		43
H	h	8	Z	z	26	,		44
I	i	9	[		27	−		45
J	j	10	£		28	.		46
K	k	11	]		29	/		47
L	l	12	↑		30	0		48
M	m	13	←		31	1		49
N	n	14	SPACE		32	2		50
O	o	15	!		33	3		51
P	p	16	"		34	4		52
Q	q	17	#		35	5		53
6		54	┌	O	79	▒		104
7		55	┐	P	80	◤	▨	105
8		56	●	Q	81	▐		106
9		57	▁	R	82	├		107
:		58	♥	S	83	▗		108
;		59	▏	T	84	└		109
<		60	╭	U	85	┐		110
=		61	╳	V	86	▂		111
>		62	○	W	87	┌		112
?		63	♣	X	88	┴		113
─		64	▕	Y	89	┬		114
♠	A	65	♦	Z	90	┤		115
│	B	66	┼		91	▌		116

Table 2.1—Continued

SET 1	SET 2	POKE	SET 1	SET 2	POKE	SET 1	SET 2	POKE
─	C	67	▒		92	▎		117
⎺	D	68	│		93	▐		118
▔	E	69	π	▚	94	▔		119
▁	F	70	◥	▨	95	▀		120
▏	G	71	SPACE		96	▃		121
▕	H	72	▌		97	┛	✓	122
╮	I	73	▄		98	▖		123
╰	J	74	▔		99	▝		124
╯	K	75	▁		100	┘		125
┗	L	76	▏		101	▘		126
╲	M	77	▒		102	▚		127
╱	N	78	▕		103			

The POKE codes given in table 2.1 provided the numbers 83 and 122 for the symbols displayed in the above exercise. These symbols are known as character set 1. Set 1 includes all of the graphics, numbers, and capital letters you normally use from the keyboard.

If you use the POKE for Set 2, then only capital and small letters, numbers, and a few other symbols are available. For most graphic oriented displays, this set is not too useful.

Reverse characters are also available with POKE statements. To get the reverse of a graphic or other symbol, just add 128 to the POKE code in table 2.1. Try the previous exercises again, but this time we will add 128 to each value.

```
POKE 1024,83+128: POKE 55296,1
POKE 2023,122+128: POKE 55295,8
```

Now you will get a reverse heart and a reverse corner symbol on the screen.

We could have added the values ourselves before writing the pokes. In that case, the pokes would be as follows:

```
POKE 1024,211: POKE 55296,1
POKE 2023,250: POKE 55295,8
```

IMPROVED SCREEN AND COLOR ADDRESSING

The addresses 1024 and 55296 are admittedly a bit difficult to work with. An improved method is to think of both the screen memory and the color memory as having 1,000 positions numbered from 0 to 999. This arrangement is shown in figure 2.2.

When this numbering is used, it must be converted to actual addresses before a value can be poked to the screen or color memory. For this purpose, the variables SC (screen) and CO (color) which have

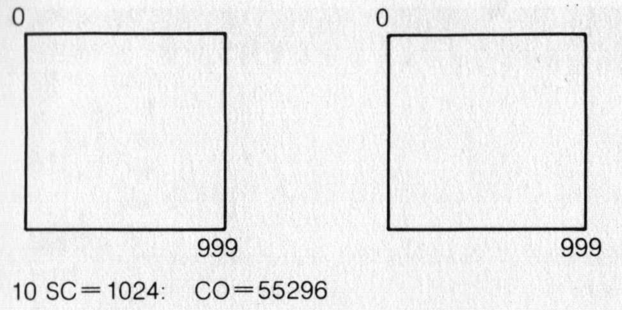


Figure 2.2 1000-Position Screen Addressing

been assigned the value of the starting address are used. The adjusted address will then be a variable, such as I, which contains a value from 0 to 999, plus the variable for screen and color as follows:

```
100 POKE I+SC,83:POKE I+CO,1
```

DRAWING A HORIZONTAL LINE

Try the following short program to draw a horizontal line across the top of the screen.

```
1 SC=1024:CO=55296
2 FOR I=0 TO 39
3 POKE I+SC,67:POKE I+CO,1
4 NEXT I
```

Notice first, the use of variables SC and CO which define the addresses of the screen and color memory. Using variables reduces the need to define these numbers more than once in the program.

The FOR statement causes the pokes to refer to the first 40 positions on the screen. The second line would be referred to as positions 40 to 79 and so on.

The pokes refer to the starting address on the screen (SC and CO) plus the value of I for each pass through the loop. The screen memory is poked with a value of 67 representing a horizontal line, and color memory is poked with a 1 to get a white line.

DRAWING A VERTICAL LINE

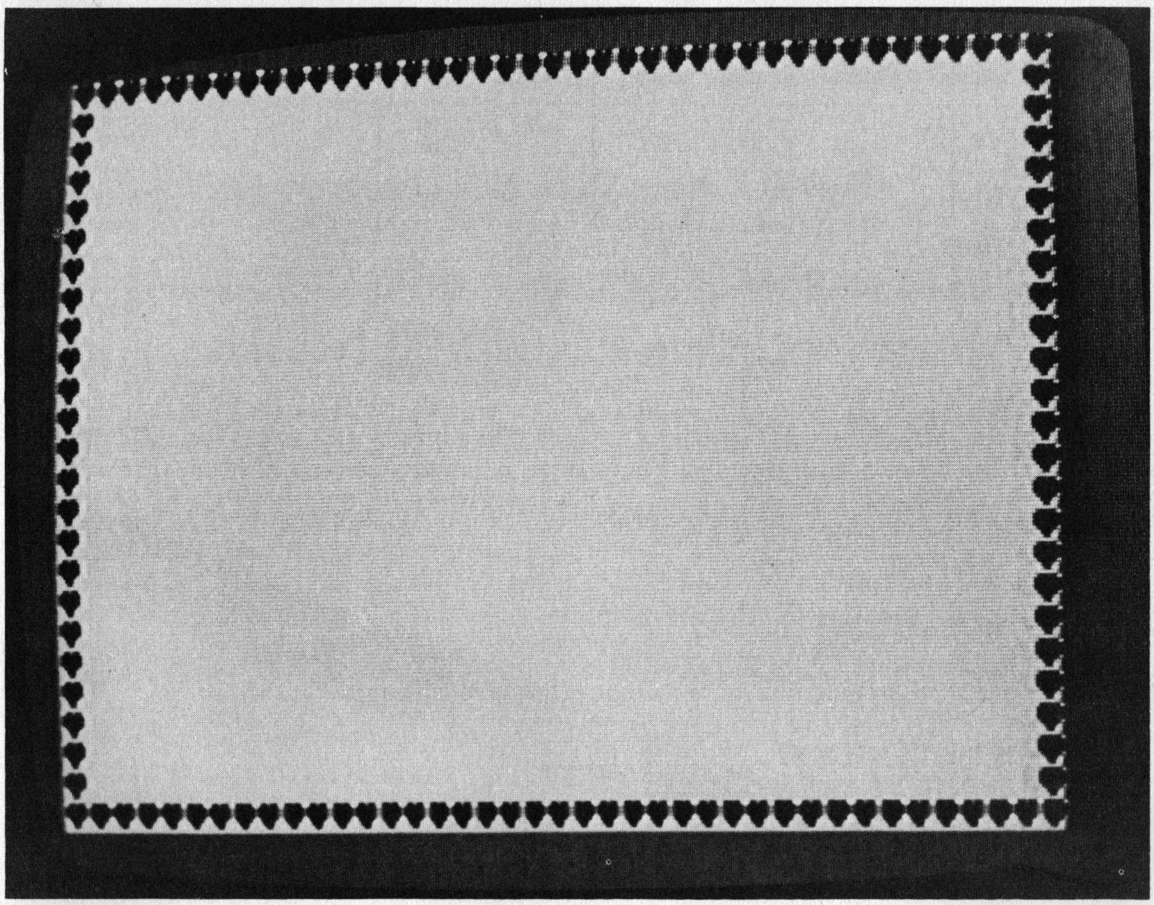
A vertical line can be drawn using a similar method; but since each character will be one complete line below the last character, the increment will need to be 40, since there are 40 positions on a line.

Try running this program.

```
1 SC=1024:CO=55296
2 FOR I=0 TO 999 STEP 40
3 POKE I+SC,101:POKE I+CO,1
4 NEXT I
```

DRAWING A BORDER

Now we are ready to put together the concepts of drawing horizontal and vertical lines into a program to draw a border of red hearts around a white screen with a green border. Here is the program.



```
10 REM BORDER
20 PRINT CHR$(147)
30 POKE 53280,5:POKE 53281,1
40 SC=1024:CO=55296
50 FOR I=0 TO 39:        REM TOP & BOTTOM
60 POKE I+SC,83:POKE I+CO,2
70 POKE I+SC+960,83:POKE I+CO+960,2
80 NEXT I
90 FOR I=0 TO 999 STEP 40:        REM LEFT & RIGHT
100 POKE I+SC,83:POKE I+CO,2
110 POKE I+SC+39,83:POKE I+CO+39,2
120 NEXT I
```

Comments on the Program

Lines	
20	Clears the screen.
30	Sets border to green and screen to white.
40	Stores screen and color addresses.
50–80	Loop for poking horizontal lines at the top and bottom of the screen.
60	Pokes top line of hearts.
70	Pokes bottom line of hearts.
90–120	Loop for poking vertical lines at the left and right sides of the screen.
100	Pokes line of hearts on the left of the screen.
110	Pokes line of hearts at the right of the screen.

Challenge

How about making this screen display into a real valentine? Can you draw a large heart made of small hearts in the center of the screen? How about a "Be My Valentine" message as well!

LUNAR LANDER: COMBINING TWO IMAGES ON THE SCREEN

Graphic images seldom exist on the screen by themselves. Usually the image is shown in the context of other information or other graphics. Both images can easily be placed on the screen by simply poking one image and then the other.

A second technique, used in the Lunar Lander example, is the designing of the main image, the lander. For simple objects, it may be possible to conceive the graphic mentally and then write the pokes to place it on the screen. However, for more complex drawings, this method is not very effective. Instead of mental gymnastics, use some graph paper as shown in figure 2.3. Here a drawing of the Lunar Excursion Module (LEM) is penciled on the paper. When the drawing is being made be sure to keep in mind the symbols that are available to you. Reverse symbols are also useful and are used extensively in this drawing.

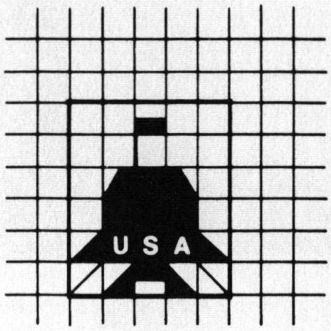


Figure 2.3 Lunar Lander Working Drawing

Now that the working drawing is complete, the next step is to determine the POKE codes to use for this image. Write down each code, line by line, in the same order that the characters appear on the design. Remember that some of these codes will be reverse characters (POKE code + 128).

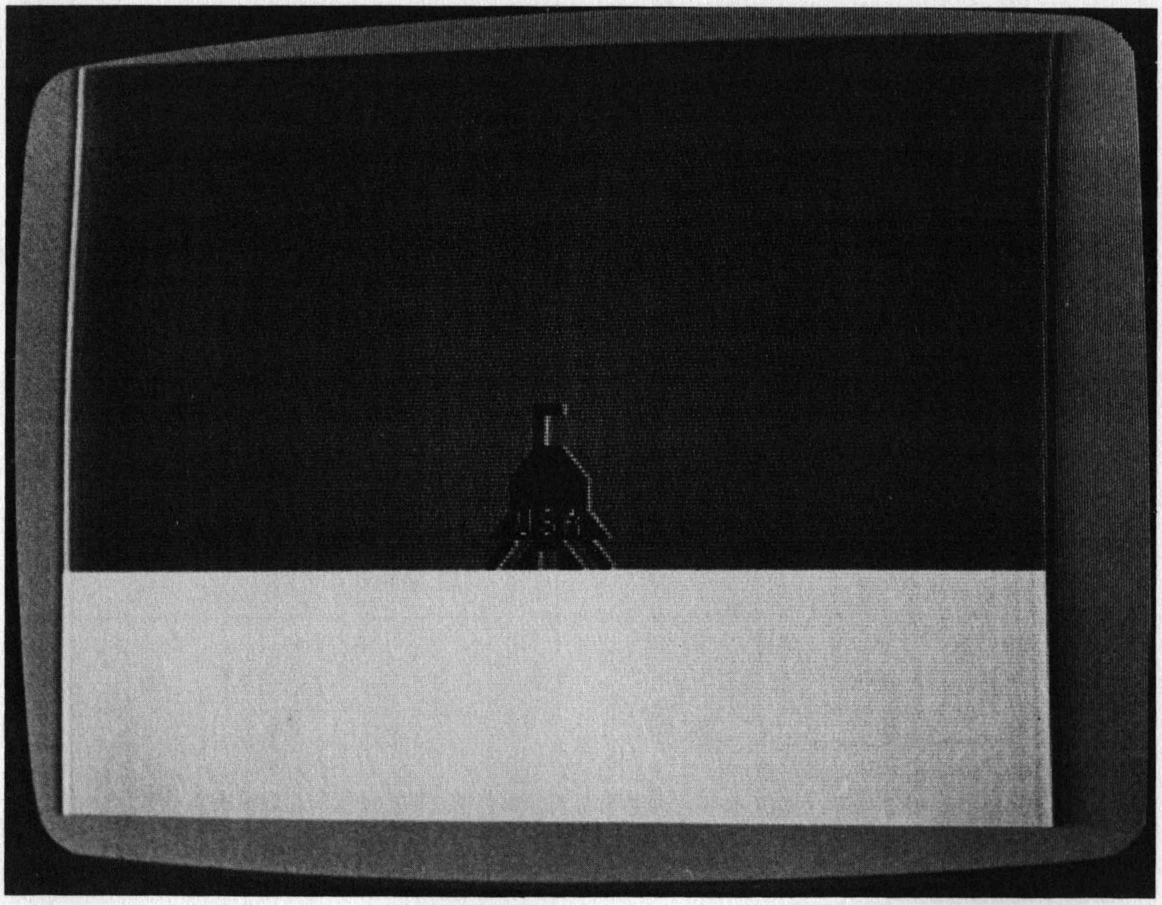
Notice that the graphic is coded as a square. Some of the screen positions are blank, so they are represented by the code 32. Using this pattern makes the operation of poking the numbers to the screen much easier than if we attempted to poke only those values representing the LEM itself.

Now we have a chart as follows:

32	32	121	32	32
32	32	101	32	32
32	233	160	223	32
32	160	160	160	32
233	149	147	129	223
78	233	120	223	77

These values must be poked to the screen to create the shape of the LEM. We could use 30 POKEs, which of course is a poor solution, or we could use a loop to read these values and poke them to the screen.

After the Lunar Lander has been poked to the screen, it might be nice to create a moonscape for it to sit on. To do this, another loop is used to poke a reverse space to the lines of the screen below the LEM. A contrasting shade of gray is used for the moonscape.



```
100 REM LUNAR LANDER
110 POKE 53280,6:POKE 53281,12
120 SC=1024:CO=55296
130 PRINT CHR$(147)
140 FOR I=458 TO 658 STEP 40
150 FOR K=I TO I+4:          REM DRAW LANDER
160 READ N
170 POKE SC+K,N:POKE CO+K,11
180 NEXT K
190 NEXT I
200 FOR I=680 TO 999:        REM MOONSCAPE
210 POKE SC+I,224:POKE CO+I,15
220 NEXT I
230 STOP
240 DATA 32,32,121,32,32
250 DATA 32,32,101,32,32
260 DATA 32,233,160,223,32
270 DATA 32,160,160,160,32
280 DATA 233,149,147,129,223
290 DATA 78,233,120,223,77
```

Comments on the Program

Lines	
110	Sets a blue border and medium gray background.
120	Sets the screen and color memory addresses.

140–190	Set the starting address of each line.
150–180	Loop to read and poke LEM codes to the screen.
160	Reads poke code.
170	Pokes the code to the screen using a dark gray color.
200–220	Loop to display moonscape.
210	Pokes reverse space with light gray for the moonscape.
240–290	Data for lander poke codes.

Challenge

Add a flag on the surface of the moon. Why not include a few craters as well? If you would like a real challenge try a new program which shows the space shuttle Columbia about to touch down on the runway.

USING PEEK AND POKE

While the POKE is used to place a value at a location on the screen, the PEEK command can be used to detect the current value of a screen memory location. This technique is used in the following program of a Christmas tree.

The poke is used to create the tree with reverse green spaces. But what is a tree without lights? Even better, flashing lights? So the peek and poke are used to turn the lights on and off to create a blinking effect.

Figure 2.4 shows how the tree is initially designed. The variable I is used to identify the top center point on the screen, where the top of the tree will display. Variable A produces an offsetting value, which determines the number of positions the tree will expand to the left and right of the central point. A starts at 1 and will increase to 16 for a total width of 32 characters at the bottom of the tree.

To blink the lights on the tree, a random function is used first to determine the location where a light will appear. If the light is already turned on at that location, determined by a peek, the program will turn it off. If the light is off, then it will be turned on. By randomizing this operation the lights blink on and off with no definite pattern.

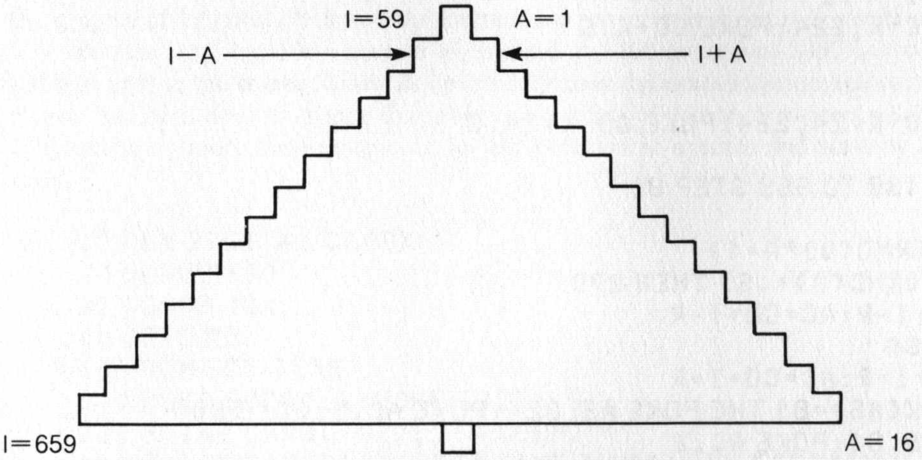
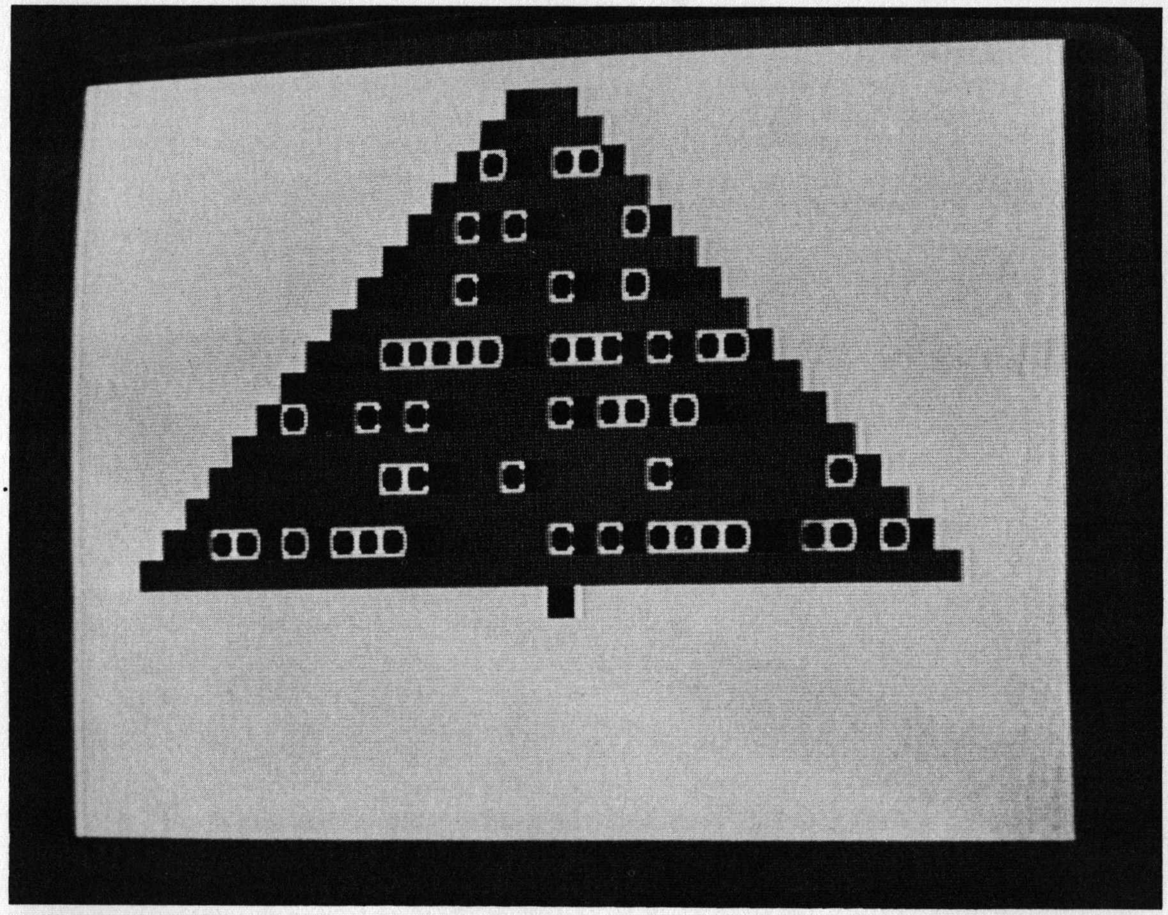


Figure 2.4 Christmas Tree Design

```
100 REM CHRISTMAS TREE
105 PRINT CHR$(147)
110 POKE 53280,5:POKE 53281,1
120 SC=1024:CO=55296
130 FOR I=59 TO 659 STEP 40
```



```
140 A=A+1
150 FOR K=I-A TO I+A
160 POKE SC+K,224:POKE CO+K,5
170 NEXT K
180 NEXT I
190 POKE SC+K+24,224:POKE CO+K+24,9
200 A=0
210 FOR I=139 TO 659 STEP 80
220 A=A+2
230 R=INT(RND(0)*A+1)
240 IF INT(RND(0)+.5) THEN 270
250 AS=SC+I-R:AC=CO+I-R
260 GOTO 280
270 AS=SC+I+R:AC=CO+I+R
280 IF PEEK(AS)=81 THE POKE AS,224:POKE AC,5:GOTO 300
290 POKE AS,81:POKE AC,2
295 FOR K=1 TO 50:NEXT K
300 NEXT I
310 GOTO 200
```

Comments on the Program

Lines	
110	Sets border to green and background to white.
120	Sets screen and color memory addresses.
130–180	Set midpoint of tree for 16 lines.

140	Increases A for the offset of each line.
150–170	Loop to poke tree +A and −A positions from the midpoint.
160	Pokes reverse spaces (224) in green (5).
190	Creates a brown (9) trunk.
200	Restarts A's value for lights.
210–300	Loop to turn lights on and off on alternate lines.
220	Sets A to establish distance from the midpoint.
230	Randomizes the position of the light on the line.
240	Chooses alternate sides of the tree.
250	Sets screen (AS) and color (AC) address on the left side of the tree.
270	Sets screen (AS) and color (AC) address on the right side of the tree.
280	If light is on (peek value 81) turns it off with a poke 224 (reverse space).
290	Light is off so pokes it on with a value of 81 (solid circle) and color red (code 2).
295	Time delay.
310	Resets A and loop back to continue blinking the lights.

Challenge

Can you place a star at the top of the tree and some gifts under it? Now make the lights multicolored by selecting different pokes with a variety of color codes. You will need to consider colors that go best on a green tree.

USING PRINTS AND POKES IN A DIGITAL CLOCK

Now that you have tried both print and pokes for creating graphics, you will be aware that for some graphics the print is the easier method. For others the poke is preferred. Of course, some of the graphics in this chapter could have used a print instead of the poke.

For example, the Christmas tree might have been easier to display using print commands. But blinking the lights would be awkward to handle with print statements, and so the poke was a superior method.

In the following program, prints and pokes are combined using the best features of each. To create the graphic of a digital clock and the instructions for using it, print statements are used.

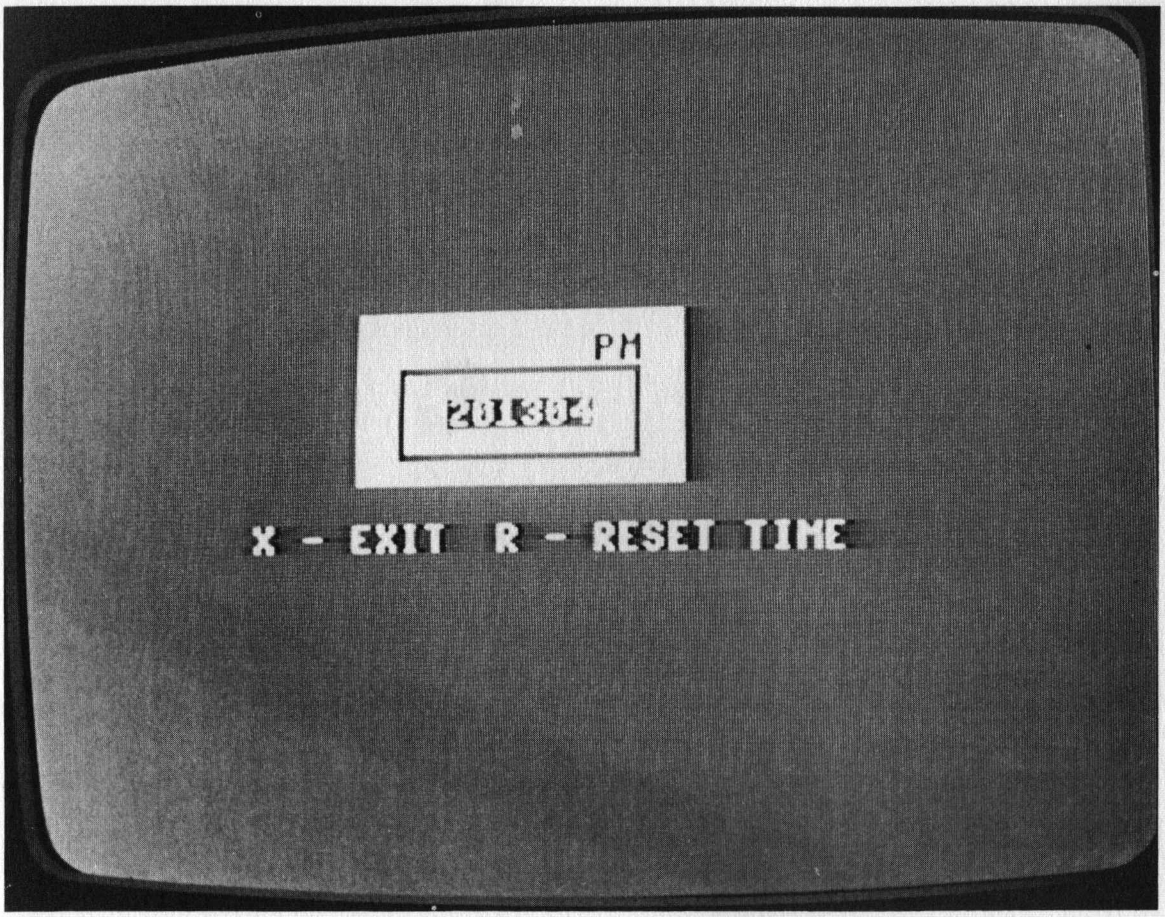
But the AM and PM indicator is changed at 1200 hours and 2400 hours by simply poking the letter A or P to the screen. Since this poke addresses the exact location, the result is instantaneous. If a print had been used to change this indicator, a string of cursor characters would be necessary, which is considerably more time-consuming for the program to execute and actually a lot more difficult to code.

```
10 REM DIGITAL CLOCK
20 GOSUB 100
30 GOSUB 400
40 GOTO 20
100 REM SET TIME
110 POKE 53280,6:POKE 53281,6
120 PRINT CHR$(147);"{4 DOWN}"
130 PRINT TAB(5);"ENTER TIME IN 24 HOUR FORMAT"
140 PRINT TAB(17);"HHMMSS"
150 PRINT:PRINT "EXAMPLE:"
160 PRINT:PRINT "8:00 AM WOULD BE ENTERED AS 080000"
170 PRINT:PRINT "1:15 PM IS ENTERED AS 131500"
180 PRINT
190 INPUT "TIME";T$
200 TI$=T$
210 RETURN
```

```
400 REM DISPLAY CLOCK
410 PRINT CHR$(147);"{7 DOWN}"
420 PRINT TAB(12);"{WHT} {RVS} O [<YYYYYYYYYYYY>] P"
430 PRINT TAB(12);"{RVS} [<H>] {10 SPACES} AM {SPACE}[<N>]"
440 PRINT TAB(12);"{RVS} [<H>] {SPACE} O [<YYYYYYYY>]
    P {SPACE} [<N>]"
450 PRINT TAB(12);"{RVS} [<H>] {SPACE} [<H>] {8 SPACES} [<N>]
    {SPACE} [<N>]"
460 PRINT TAB(12);"{RVS} [<H>] {SPACE} L [<PPPPPPPP>] @
    {SPACE} [<N>]"
470 PRINT TAB(12);"{RVS} L [<PPPPPPPPPP>]@"
480 PRINT
490 PRINT TAB(8);"X - EXIT R - RESET TIME"
500 PRINT CHR$(19);"{11 DOWN}"; TAB(16); TI$
510 IF LEFT$(TI$,2)>="12" THEN POKE 1406,144:GOTO 530
520 POKE 1406,129
530 GET A$
540 IF A$="" THEN 500
550 IF A$="X" THEN STOP
560 IF A$="R" THEN RETURN
570 GOTO 500
```

Comments on the Program

Lines	
20	Go to set time subroutine.
30	Go to display time subroutine.
100–210	Set time subroutine.
110	Sets border and screen to blue (6).
120	Clears screen and cursor down 4 lines.
130–180	Print operating instructions.
190	Inputs current time (T$) in 24–hour format.
200	Assigns time (T$) to time function (TI$).
400–570	Display clock and time subroutine.
410	Clears screen and cursor down 7 lines.
420–480	Display clock graphic.
490	Prints exit and clock reset message.
500	Sends cursor home (CHR$(19)) and positions it on the screen before displaying time (TI$).
510	If the hour is 12 noon or later, pokes a P to the AM–PM indicator.
520	AM time so pokes an A.
530	Gets response to exit and clock reset message,
540	No response to message so continues loop to display time.
550	"X" response means to terminate program.
560	"R" response means to return to the beginning and ask for a new time.



```
                PM
        201304

X - EXIT  R - RESET TIME
```

Challenge

Although the AM and PM indicators were written into the program with reverse characters, they give the appearance on the screen of normal nonreversed letters. This is because of the color combinations chosen. Revise the program so that AM and PM will appear reversed on the clock.

The clock currently displays in 24–hours time. That is due to the way the TI$ function works in the Commodore 64. Can you devise a way to change this value from TI$ so that it can be displayed in a 12–hour format on the clock?

This program has lots of possibilities! Why not add a third message under the clock giving the option "S" for stopwatch. When S is pressed, go to another subroutine that uses the TI function and provides for starting and stopping the counter.

3

Animation

Like me, you have probably admired some of the microcomputer graphic programs or video games that are enhanced by animation to make them more interesting. In this chapter, we will look at some of the basic techniques for creating animation in a BASIC program. Since we are using BASIC, the methods used will be somewhat limited, but still effective, for many programs. For more advanced graphics, the next few chapters will explain how sprite graphics may be used for even more spectacular results.

PRINCIPLES OF ANIMATION

If you have seen how TV or movie cartoons are made, you know that the action is created by drawing an image, such as Mickey Mouse, many times with a slight change in position for each image. By displaying these images in quick succession, the illusion of movement is created.

This cartoon technique is normally not used on the computer for two very good reasons. One, it would require too much memory to store all of these different images; and, two, except for elementary types of graphics, BASIC is too slow to display many images rapidly. However, we will be using a variation of the cartoon technique to create animation on the Commodore 64.

One way to produce animation is to print an object on the screen and then print it again and again but in a slightly different position. This is quite similar to the concept of displaying many images in quick succession. The main difference in programming is that it is the same object that is displayed many times, instead of a series of slightly different objects.

ANIMATING WITH PRINT STATEMENTS

As we saw in chapter 1, a graphic could be easily produced on the screen by using a print statement. Graphics may also be animated by placing the print statement or statements inside a loop in the program.

The main advantage of using the print for producing animation is that it is generally quite easy to read the program and understand what it is doing. That means it is also easy to write the program to produce the animation you desire.

Disadvantages? Well, for rather small images, the animation will be relatively smooth, but the larger you make the graphic image, the less smooth the animation. Speed can also become a problem with larger images. However, for smaller ones you may very well have to slow things down.

HORIZONTAL MOVEMENT: LEFT TO RIGHT

To animate a small object, let's just use an asterisk (*) for now, a very simple method is to place the print statement inside a FOR loop and print the character so that it moves across the line. Try this short program:

```
10 FOR I=1 TO 39
20 PRINT "*";
30 NEXT I
```

If you run this program the output on the screen is a line of asterisks like this:

```
***************************************
```

What went wrong? Instead of the asterisk moving across the screen, a line of asterisks is printed across the screen. This problem points out a very elementary principle of computer animation. **Each time an object or graphic is displayed, the previous object or graphic must be erased.** In other words, when the asterisk is printed, the previous asterisk must be wiped out.

Now try this revised program.

```
10 FOR I=1 TO 39
20 PRINT "{SPACE}*{LEFT}";
30 NEXT I
```

Statement 20 has been revised. How does this work? When the asterisk is first printed, the cursor left will move the cursor position back to where the asterisk was displayed. The second time through the loop, the space prints on top of the previous asterisk, wiping it out, and the new asterisk is printed one position to the right, giving the illusion of movement. This process continues 39 times through the loop.

When you run this program, you will no doubt feel that the asterisk is moving too fast. To slow things down to the right speed use a delay loop. Of course, what is right in this example is pretty much a personal decision. I've used 50 in the delay loop in the next program. If this is either too slow or too fast for you, try varying the value of the delay loop in statement 25, until you get the speed you like.

```
10 FOR I=1 TO 39
20 PRINT "{SPACE}*{LEFT}";
25 FOR D=1 TO 50:NEXT D
30 NEXT I
```

Comments on the Program

Lines	
10–30	Loop to animate object across 39 columns.
20	Erases the last asterisk and prints the next one.
25	Delay loop.

HORIZONTAL MOVEMENT: RIGHT TO LEFT

The previous program used the natural direction of the print statement to produce left to right movement. To animate an object in the opposite direction, we have two choices. One, the print could use cursor characters to position the cursor to the right of the screen, or the second choice would be to use the TAB function to set the initial cursor position.

TAB is obviously the better choice. Just tell it the position on the line that you want and "Voila!" the character will print there.

Here is a program to move the asterisk from right to left:

```
10 PRINT TAB(38);
20 FOR I=1 TO 36
30 PRINT "{3 LEFT}*{SPACE}";
40 FOR D=1 TO 50:NEXT D
50 NEXT I
```

Comments on the Program

Lines	
10	Tabs to column 38 the initial starting position.
20–50	Loop to move the object 36 positions from right to left.
30	Moves cursor 3 columns left, prints the asterisk, and erases the last one.
40	Delay loop.

VERTICAL MOVEMENT: TOP TO BOTTOM

The next technique is to move an object from the top of the screen to the bottom, such as for dropping a projectile or a parachute. For convenience, we will use the asterisk once again in the following program:

```
10 PRINT CHR$(147);
20 FOR I=1 TO 22
30 PRINT TAB(20);"*"
40 FOR D=1 TO 50:NEXT D
50 PRINT TAB(20);"{UP}{SPACE}"
60 NEXT I
```

Comments on the Program

Lines	
10	Clears screen.
20–60	Loop to drop the object from the top of the screen down 22 lines.
30	Tabs to column 20, approximately the center of the screen and prints the asterisk.
40	Time delay loop.
50	Tabs again to column 20, moves up to the previous line, and prints a space to erase the asterisk.

VERTICAL MOVEMENT: BOTTOM TO TOP

Moving an object from the bottom of the screen to the top is not quite as easy as the other direction. But let's "think positive"; it can be done!

The key to movement in this direction is to use cursor control characters to position the cursor each time the object is about to be displayed. In the following program, a variable D$ is used to provide the cursor down characters, while the LEFT$ function is used in the print statement to select the correct number of cursor characters to move to the exact line on the screen. Since cursor movement must always start from a known position, home is used [CHR$(19)] to establish this location for each pass through the loop.

```
10 PRINT CHR$(147)
20 D$="{22 DOWN}"
30 FOR I=1 TO 21
40 PRINT CHR$(19);
```

```
50 PRINT LEFT$(D$,22-I);
60 PRINT TAB(20);"*"
70 FOR D=1 TO 40:NEXT D
80 PRINT TAB(20);"{UP}{SPACE}"
90 NEXT I
```

Comments on the Program

Lines	
10	Clears screen.
20	Assigns 22 cursor down characters to D$.
30–90	Loop to display object.
40	Sends the cursor home.
50	Positions cursor. When I is 1 the cursor will be set down 21 lines (22−1=21).
60	Prints asterisk at column 20.
70	Time delay.
80	Erases the object before the next animation loop.

SATURN V ROCKET

A favorite with everyone is creating a rocket blasting off from the launching pad. This program applies the basic technique of moving an object vertically from the bottom to the top of the screen. As you will recall, this method used cursor down characters to position the object at the bottom of the screen. Subsequent print statements moved the object up a line, until it reached the top of the screen.

A new problem that the rocket program presents is that the object (the rocket) to be moved consists of many more characters than did the asterisk in the previous program. This is easily solved by simply using a series of print statements to print the graphic as we want it to appear.

But, it is also necessary to erase the graphic from the screen each time, so that we don't get multiple images. Instead of trying to erase the entire rocket each time just use one extra print at the end of the statements that print the rocket. This extra statement will print blanks to erase the last line of the previous rocket. What about the remainder of the previous rocket, you ask? Well, that is taken care of by printing the next rocket. Since the second rocket overprints the first, one line up, all of the first rocket will be erased except for the last line. Thus the need for the extra print with a blank line.

Figure 3.1 shows how the rocket is lifted off the pad. Since the rocket consists of 7 lines of graphic characters, the control loop cannot use all lines of the screen. Instead, only 15 lines are used. So when the rocket is printed, it is first positioned 15-I lines down the screen. This location (line 14 the first time) will leave enough lines to display the entire rocket at the bottom of the screen without causing scrolling to occur.

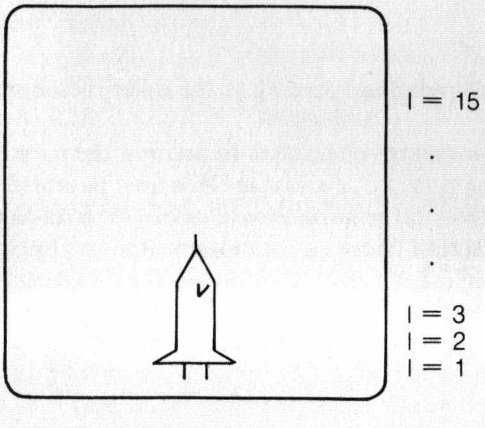
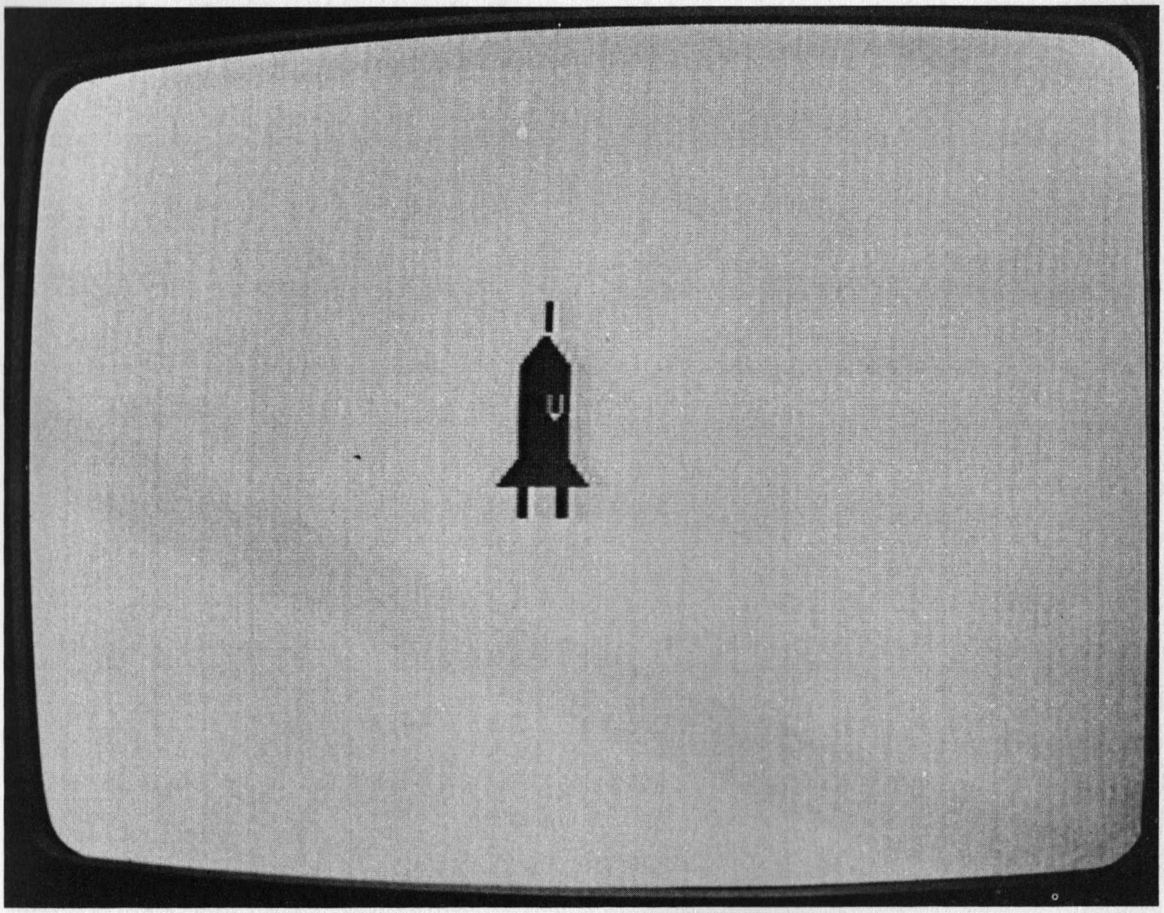


Figure 3.1 Controlling the Rocket Animation with Variable I.



```
100 REM ROCKET
110 POKE 53280,3:POKE 53281,3
120 A$="{14 DOWN}"
130 PRINT CHR$(147)
140 FOR I=1 TO 15
150 PRINT CHR$(19); "{BLU}"
160 PRINT LEFT$(A$,15-I);
170 PRINT TAB(16); "{4 SPACES}[<G>]{3 SPACES}"
180 PRINT TAB(16); "{3 SPACES}{RVS} }£ [<*>]{OFF}{2 SPACES}"
190 PRINT TAB(16); "{3 SPACES}{RVS}{2 SPACES}{OFF}{2 SPACES}"
200 PRINT TAB(16); "{3 SPACES}{RVS}{SPACE} V{OFF}{SPACE}"
210 PRINT TAB(16); "{3 SPACES}{RVS}{2 SPACES}{OFF}{2 SPACES}"
220 PRINT TAB(16); "{2 SPACES}{RVS}£ {2 SPACES}[<*>]{OFF}
    {SPACE}"
230 PRINT TAB(16); "{3 SPACES}[<J>][<L>]{3 SPACES}"
240 PRINT TAB(16); "{10 SPACES}"
250 NEXT
```

ANIMATING MULTIPLE OBJECTS: THE TRAIN

Graphic programs frequently need to animate two or more different objects and often in different directions. For example, imagine an old steam engine chugging across the screen from right to left, while white smoke is puffing from its stack and floating off to the right into the breeze.

To produce this action, a loop is needed to move the train from right to left. The following code homes the cursor each time through the loop and then tabs to a column each time to display the letter T for train.

```
10 FOR I=38 TO 1 STEP -1
20 PRINT CHR$(19)
30 PRINT TAB(I) "T{SPACE}"
40 FOR K=1 TO 100:NEXT
50 NEXT I
```

If you run this program, you will see the letter T make its passage across the screen from right to left. Using home resets the cursor to the beginning of the screen so that the TAB will reposition each time to the same line. The value of I will be reduced by one every time through the loop, so that the letter will print one position to the left of the last time.

Now, the next problem is to move some smoke through the air but from left to right. We'll let S represent the smoke for now. So replace statement 40 with these new lines:

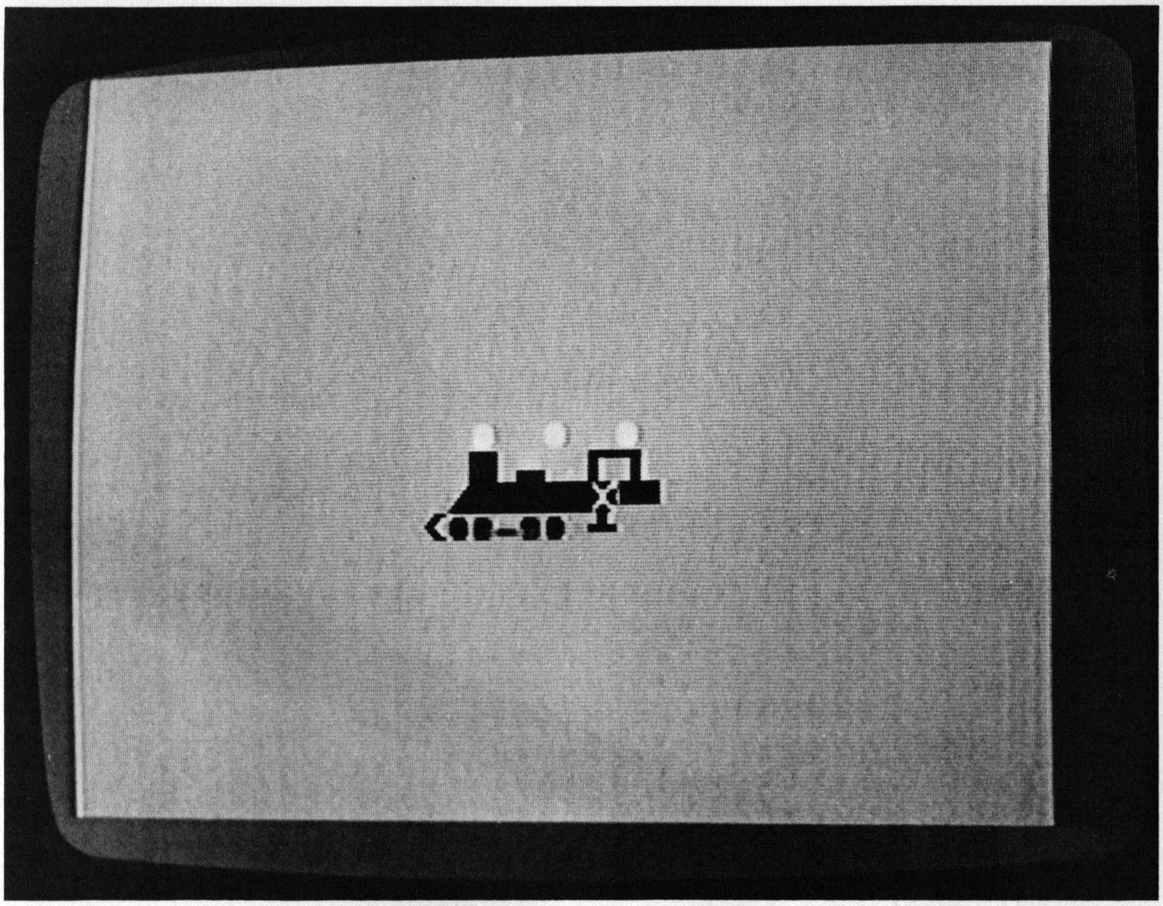
```
40 FOR J=1 TO 3
42 PRINT TAB(I+J)"S";
44 FOR K=1 TO 40:NEXT
46 PRINT "{LEFT}{SPACE}";
48 NEXT
```

What is happening in this code is that the letter S prints 1 position to the right of the T (I+J). After the delay loop of line 44, the print at 46 moves the cursor left one position and erases the S. Then the J loop is repeated a second and third time moving the S one position further to the right every time. During this process, the T has not been moved.

Now let's apply these techniques to the original objective of producing a train with smoke coming from its stack. The following program displays a black (what else?) steam engine with red wheels even, chugging across the screen (light gray background) with white puffs of smoke leaving the stack. Just what we wanted!

This program uses some cursor down characters to position the train down in the bottom half of the screen. Otherwise, the method is essentially the same as used above for the letters T and S.

```
100 REM TRAIN
110 POKE 53280,6:POKE 53281,15
120 A$="{12 DOWN}"
130 PRINT CHR$(147)
140 FOR I=28 TO 1 STEP -1
150 PRINT CHR$(19); A$
160 PRINT TAB(I)"{BLK}{2 SPACES}{RVS}{SPACE}{OFF}
    {SPACE}[<O>]{2 SPACES}OP{SPACE}"
170 PRINT TAB(I) "{SPACE}{RVS}£{5 SPACES}VL@{OFF}{SPACE}"
180 PRINT TAB(I)"<{RED}QQFQQ{BLK}{SPACE}[<E>]{SPACE}"
190 PRINT CHR$(19); A$; TAB(I+2);
200 FOR J=1 TO 3
210 PRINT TAB(J); "{WHT}QW{BLU}";
220 FOR K=1 TO 40:NEXT
230 PRINT "{2 LEFT}{2 SPACES}";
240 NEXT
250 NEXT I
260 PRINT CHR$(19)
```



Comments on the Program

Lines	
110	Sets light gray screen and blue border.
120	Sets up a string of cursor downs.
130	Clears screen.
140–250	Loop to controlling the train's movement across the screen from right to left.
150	Sends cursor home and moves down 12 lines.
160–180	Print the train.
190	Positions cursor for the smoke.
200–240	Loop to animate the smoke from the stack.
210	Displays the white puffs of smoke.
220	Delay loop.
230	Erases the puffs of smoke.
260	Sends cursor home so that the READY prompt will not wipe out the train.

Challenge

Can you control the smoke so that it will rise several lines above the train? How about a track for the train to run on. You could also add a coal car, but remember, if the entire train becomes too long, it will not be a very effective animation.

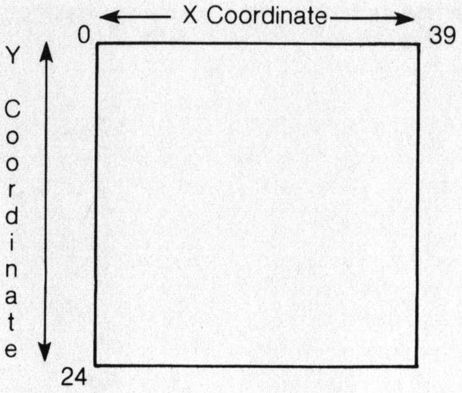


Figure 3.2 X and Y Coordinates for Screen Memory

ANIMATING WITH POKE STATEMENTS

We saw in chapter 2 how graphics could be created by using poke statements. Using similar methods, it is possible to create animation on the screen with poke statements. For some types of animated graphics, such as games, the poke procedure can be very effective.

When using pokes, it is often helpful to think of the screen as being formed with X and Y coordinates as shown in figure 3.2. The X coordinate refers to the 40 columns across the screen and is numbered from 0 to 39. The Y coordinate represents the number of lines on the screen; on the Commodore 64 there are 25 lines and these are numbered 0 to 24.

Here is a formula that is helpful when addressing the screen. Remember that the screen memory starts at address 1024.

Screen address = 1024 + X + 40 * Y

By inserting the values of the X and Y coordinates, the formula will then give the exact screen address for that location.

A similar formula can be used for color memory.

Color address = 55296 + X + 40 * Y

The following code gives a useful technique for determining the screen and color addresses to poke in a program. The variables SC and CO refer to screen and color addresses that we have used before. The new variables, PS and PC, refer to the addresses needed to poke the screen memory (PS) and the color memory (PC).

```
10 SC=1024: CO=55296
20 PS=SC+X+40*Y
30 PC=CO+X+40*Y
```

To use this code, variables X and Y need to be given values which refer to the coordinates of the screen. Then PS and PC can be used directly in a poke statement. This technique is used in the following four programs to move a yellow asterisk both horizontally and vertically.

HORIZONTAL MOVEMENT: POKING LEFT TO RIGHT

Creating movement horizontally with pokes is a matter of holding the Y axis constant and varying the X axis. In the following program the Y axis is set at 12, about halfway down the screen, and the X axis is varied from 0 to 39. Of course, animating does not need to operate at line 12 but could occur at any line you choose.

```
100 REM MOVE LEFT TO RIGHT
110 SC=1024:CO=55296
120 POKE 53280,6:POKE 53281,0
130 PRINT CHR$(147)
140 Y=12
150 FOR X=0 TO 39
160 PS=SC+X+40*Y
170 PC=CO+X+40*Y
180 POKE PS,42
190 POKE PC,7
200 FOR D=1 TO 50:NEXT D
210 POKE PS,32
220 POKE PC=0
230 NEXT X
```

Comments on the Program

Lines	
110	Sets screen and color memory addresses.
120	Blue border and black screen.
130	Clears screen.
140	Sets Y axis to line 12.
150–230	Vary X axis from column 0 to 39.
160	Sets screen address to poke.
170	Sets color address to poke.
180	Pokes asterisk (42) to screen.
190	Pokes yellow (7) color to memory.
200	Delay loop.
210	Pokes space (32) to erase the asterisk.
220	Sets color to black (0) again.

HORIZONTAL MOVEMENT: POKING RIGHT TO LEFT

Reversing the direction of horizontal animation is quite simple. All you need to do is vary the X axis so that it refers to the values 39 to 0 in descending order. The next program demonstrates this technique.

```
100 REM MOVE RIGHT TO LEFT
110 SC=1024:CO=55296
120 POKE 53280,6:POKE 53281,0
130 PRINT CHR$(147)
140 Y=12
150 FOR X=30 TO 0 STEP -1
160 PS=SC+X+40*Y
170 PC=CO+X+40*Y
180 POKE PS,42
190 POKE PC,7
200 FOR D=1 TO 50:NEXT D
210 POKE PS,32
220 POKE PC=0
230 NEXT X
```

Comments on the Program

Lines	
110	Sets screen and color memory addresses.
120	Blue border and black screen.
130	Clears screen.

140	Sets Y axis to line 12.
150–230	Vary X axis in the reverse direction from column 39 to column 0 by steps of −1.
160	Sets screen address to poke.
170	Sets color address to poke.
180	Pokes asterisk (42) to screen.
190	Pokes yellow (7) color to memory.
200	Delay loop.
210	Pokes space (32) to erase the asterisk.
220	Sets color to black (0) again.

VERTICAL MOVEMENT: POKING TOP TO BOTTOM

To animate an object from the top of the screen in a downward direction using pokes is a simple matter of holding the X axis constant and varying the Y axis. A variation of the previous program shows how this movement is produced.

```
100 REM MOVE TOP TO BOTTOM
110 SC=1024:CO=55296
120 POKE 53280,6:POKE 53281,0
130 PRINT CHR$(147)
140 X=19
150 FOR Y=0 TO 24
160 PS=SC+X+40*Y
170 PC=CO+X+40*Y
180 POKE PS,42
190 POKE PC,7
200 FOR D=1 TO 50:NEXT D
210 POKE PS,32
220 POKE PC=0
230 NEXT Y
```

Comments on the Program

Lines	
110	Sets screen and color memory addresses.
120	Blue border and black screen.
130	Clears screen.
140	Sets X axis to column 19.
150–230	Vary Y axis from line 0 to 24.
160	Sets screen address to poke.
170	Sets color address to poke.
180	Pokes asterisk (42) to screen.
190	Pokes yellow (7) color to memory.
200	Delay loop.
210	Pokes space (32) to erase the asterisk.
220	Sets color to black (0) again.

VERTICAL MOVEMENT: POKING BOTTOM TO TOP

Animating an object from the bottom of the screen in an upward direction using pokes is similar to the last program. Again, the program holds the X axis constant and varies the Y axis in the reverse direction from line 24 at the bottom of the screen to line 0 at the top.

```
100 REM MOVE BOTTOM TO TOP
110 SC=1024:CO=55296
120 POKE 53280,6:POKE 53281,0
130 PRINT CHR$(147)
140 X=19
150 FOR Y=24 TO 0 STEP -1
160 PS=SC+X+40*Y
170 PC=CO+X+40*Y
180 POKE PS,42
190 POKE PC,7
200 FOR D=1 TO 50:NEXT D
210 POKE PS,32
220 POKE PC=0
230 NEXT Y
```

Comments on the Program

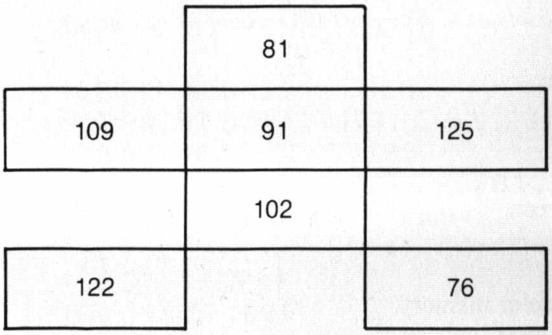
Lines	
110	Sets screen and color memory addresses.
120	Blue border and black screen.
130	Clears screen.
140	Sets X axis to column 19.
150–230	Vary Y axis from line 24 to 0 in steps of −1.
160	Sets screen address to poke.
170	Sets color address to poke.
180	Pokes asterisk (42) to screen.
190	Pokes yellow (7) color to memory.
200	Delay loop.
210	Pokes space (32) to erase the asterisk.
220	Sets color to black (0) again.

Challenge

Can you move a ball across the screen to the right and then reverse its direction? Can you write a program to move the ball diagonally? Write a program to move the ball clockwise around the screen's perimeter.

STICKMAN: ANIMATING SEVERAL CHARACTERS SIMULTANEOUSLY

When images are to be animated using pokes, it is not unusual to form the graphic with several characters. In this example, a green Martian stickman is moved across the screen horizontally from left to right. The characters forming the stickman are four lines high and three columns wide and consist of the following graphic poke characters.



The technique for poking this image on the screen is similar to those discussed earlier. But, since we are using several characters, we will choose a central point which will be the address for PS, the screen memory, and also for PC, the color memory. Then all other characters will be poked relative to this address using the following adjustments.

	81 PS−40	
109 PS−1	91 PS	125 PS+1
	102 PS+40	
122 PS+79		76 PS+81

Addressing for color memory will be the same as this diagram except that the variable will be PC instead of PS.

Animating is accomplished by varying the value of the X coordinate; however, clearing the image after each display is done with the print clear screen operation. This method is faster and uses less program code than poking each screen address with a blank code. The trade-off is reduced smoothness in the movement of our stickman. This problem is easily overcome when using sprite graphics, as we will see in the next chapter.

```
100 REM STICKMAN
110 SC=1024:CO=55296
120 POKE 53280,6:POKE 53281,0
130 PRINT CHR$(147)
140 Y=21
150 FOR X=2 TO 37
160 PS=SC+X+40*Y
170 PC=CO+X+40*Y
180 POKE PS-40,81:POKE PC-40,5
190 POKE PS-1,109:POKE PC-1,5
200 POKE PS,91:POKE PC,5
210 POKE PS+1,125:POKE PC+1,5
220 POKE PS+40,102:POKE PC+40,5
230 POKE PS+79,122:POKE PC+79,5
240 POKE PS+81,76:POKE PC+81,5
250 FOR D=1 TO 10:NEXT D
260 PRINT CHR$(147)
270 NEXT X
```

Comments on the Program

Lines	
110	Sets screen and color addresses.
120	Blue border and black screen.
130	Clears screen.
140	Sets Y coordinate to line 21.
150–270	Loop to animate across the screen.
160	Screen address.
170	Color address.
180–240	Pokes stickman to screen and color memory.
250	Time delay.
260	Clears screen.

Challenge

How about adding some other colors to stickman, such as yellow shorts? Why not add a hat? Can you devise a method for giving his legs movement as the rest of the body moves across the screen?

BOUNCING BALL

Many games involve moving an object around the screen until it hits an object, such as a paddle or a wall. In this program, we will develop the technique for moving a ball horizontally, vertically, or diagonally until it hits the inside of a rectangular wall. When the wall is hit, the ball bounces off in the reverse direction.

Figure 3.3 shows the wall and some of the variables associated with the program. As with some of the previous programs, X and Y are used to locate the ball's position on the screen. As the ball moves, its coordinates are increased by the value of DX and DY to give the ball its direction.

The variables WL, WR, WT, and WB are used to identify the position of the walls. When either X or Y's value is equal to a value of one of the walls, the program reverses the direction of either X or Y. This direction is controlled by variables DX and DY, which take on the values of either 1 or −1. Depending on which wall is hit either DX or DY will change direction. In the case of hitting a corner, when two adjacent walls are hit both DX and DY are changed.

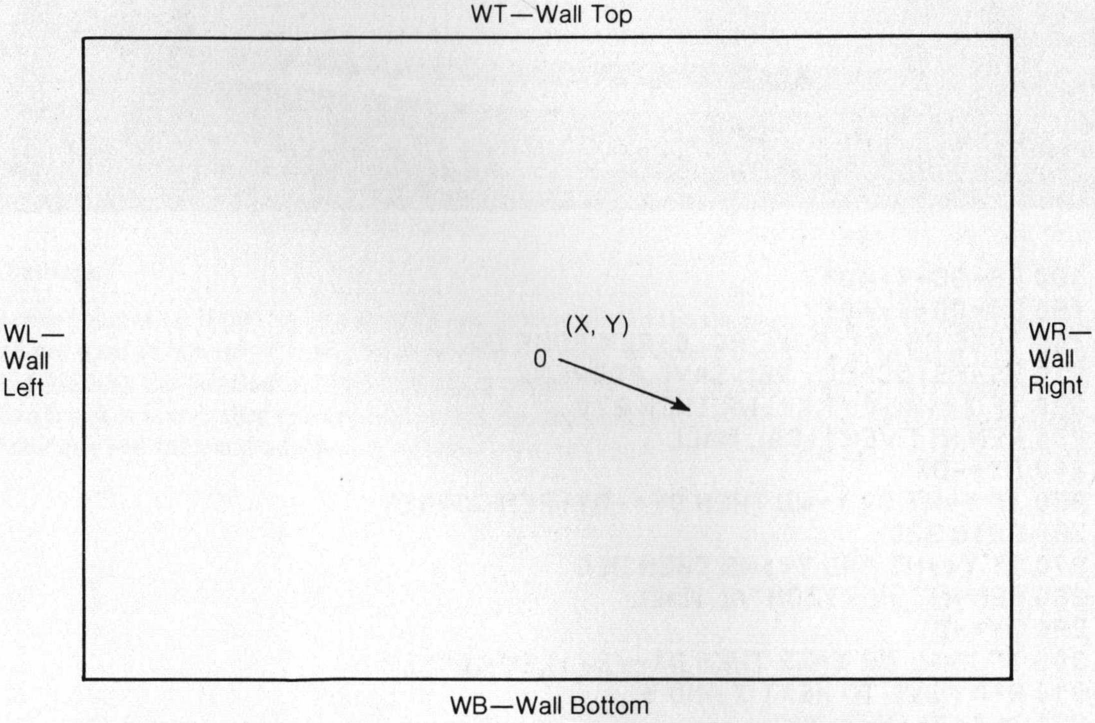
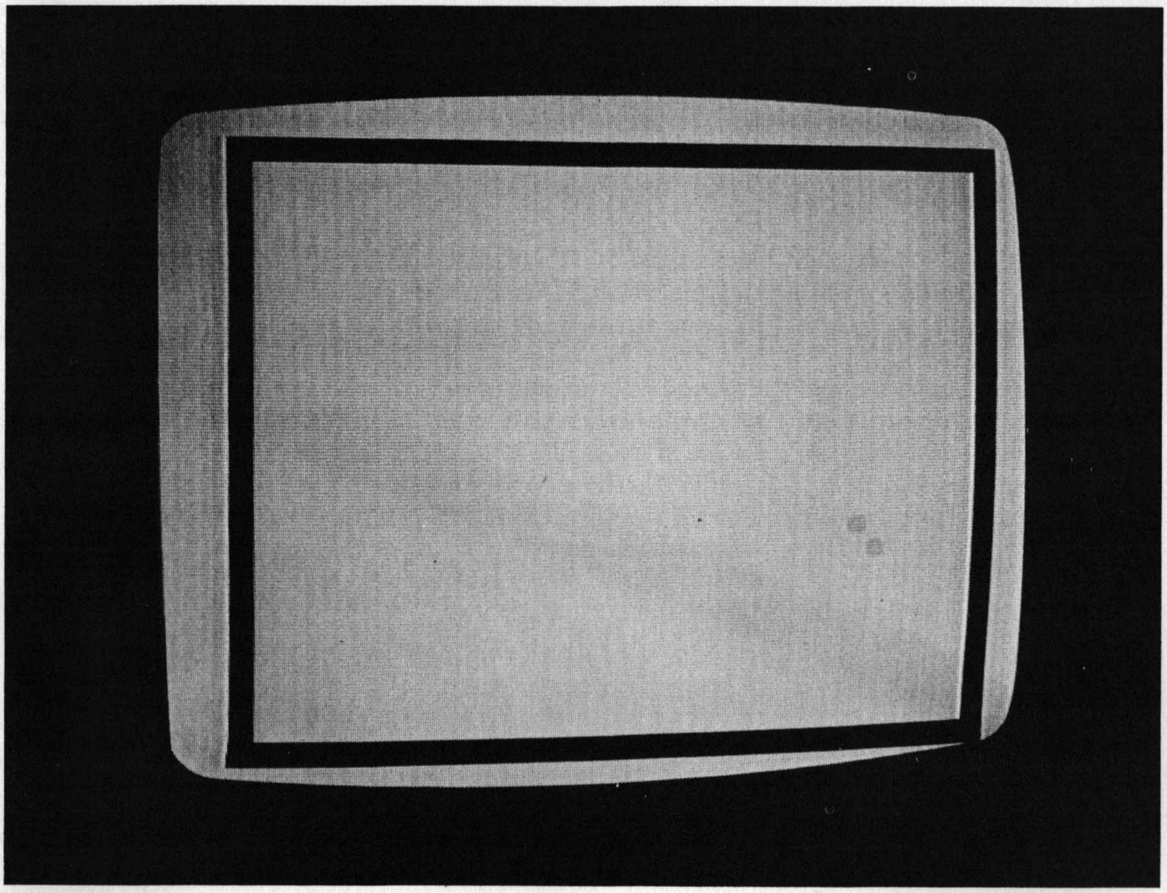


Figure 3.3 Bouncing a Ball Off Horizontal and Vertical Walls

```
100 REM BOUNCING BALL
110 PRINT CHR$(147)
120 SC=1024:CO=55296
130 POKE 53280,7:POKE 53281,7
130 WL=1:WR=38:WT=1:WB=23
140 X=2:Y=2:OS=SC+42:OC=CO+42
150 DX=1:DY=1
160 GOSUB 1000
170 POKE OS,32:POKE OC,7: REM CLEAR BALL
```



```
180 PS=SC+X+40*Y
190 PC=CO+X+40*Y
200 POKE PS,81:POKE PC,0:REM POKE BALL
210 OS=PS:OC=PC: REM SAVE ADDRESS
220 IF X<>WL AND X<>WR THEN 270
230 REM HIT VERTICAL WALL
240 DX=-DX
250 IF Y=WT OR Y=WB THEN DY=-DY:REM CORNER
260 GOTO 320
270 IF Y<>WT AND Y<>WB THEN 320
280 REM HIT HORIZONTAL WALL
290 DY=-DY
300 IF X=WL OR X=WR THEN DX=-DX:REM CORNER
310 REM MOVE TO NEXT X AND Y
320 X=X+DX:Y=Y+DY
330 GOTO 170
1000 REM DRAW WALLS
1010 FOR I=0 TO 39
1020 POKE I+SC,160:POKE I+CO,0
1030 POKE I+SC+960,160:POKE I+CO+960,0
1040 NEXT I
1050 FOR I=0 TO 999 STEP 40
1060 POKE I+SC,160:POKE I+CO,0
1070 POKE I+SC+39,160:POKE I+CO+39,0
1080 NEXT I
1090 RETURN
```

Comments on the Program

Lines	
100–120	Clears screen, store screen and color addresses, and set screen and border colors to yellow.
130	Stores X and Y coordinates of the left and right walls (WL and WR) and top and bottom walls (WT and WB).
140	Stores starting position of the ball (X and Y) and initializes old position of X and Y.
150	DX and DY establish the direction and increment for the X and Y coordinates.
160	Subroutine 1000 draws the walls.
170	Clears old ball before displaying the next one.
180–190	Store ball's poke address for screen and color.
200	Places black ball on the screen at the X and Y coordinate.
210	Saves coordinates in old screen (OS) and color (OC).
220	Checks if ball has hit the left or right wall.
240	A left or right wall was hit so reverses the direction of the X coordinate.
250	If a corner was hit, reverses the Y coordinate also.
260	Go to update X and Y values.
270	Checks if ball has hit the top or bottom wall.
290	A top or bottom wall was hit so reverses the direction of the Y coordinate.
300	If a corner was hit, reverses the direction of the X coordinate also.
320	Increases the X and Y coordinates so the ball can move one position.
330	Go to 170 to clear the old ball and display a new one in the next position defined by X and Y.
1000–1090	Subroutine to draw the walls in a rectangular shape. The technique for drawing this wall is adapted from the program in chapter 2 for drawing an outline of the screen.

Challenge

Revise subroutine 1000 to place an additional object within the space outlined by the wall. Now change the program to bounce off this object as well. One way to detect the object is to use a peek statement that looks at the position the ball will reach before the ball is actually moved. If a screen code other than blank is found, then the ball has struck the object. Try this with a single object first to master the technique and then add additional objects.

4

Sprite Graphics

The past three chapters showed a variety of ways that graphics and animation could be produced on the Commodore 64. These methods are certainly useful and have their place in many of the programs you may write for use on your computer. But, you may have also discovered some limitations with these methods.

One limitation is the shape that could be imparted to a graphic. Although the keyboard graphic symbols can be combined to produce some excellent results, no doubt you have found times when just the right character for your design was not available.

A second problem likely occurred when you wanted to animate your graphic. That problem was in attempting to get the graphic to move smoothly across the screen. When the object was small, such as the ball, this was not much of a problem, but for larger objects, smoothness became more difficult to achieve.

Well, there is an answer. Sprite graphics!

With sprite graphics, using the VIC-II chip, you will be able to create images (called sprites) made up of your own shapes instead of depending upon the shapes on the keyboard. And you will be able to define up to eight different sprites on the screen at one time. These sprites can be moved smoothly around the screen, although in BASIC their speed will be limited, especially when several sprites are moved together.

There are other advantages to sprite graphics over keyboard graphics. One is that collisions between them can be easily detected. This is a super feature, especially for games. Another advantage is that one sprite can be made to pass in front of or behind another sprite by defining sprite priorities. This feature gives a three-dimensional quality to your graphics which cannot be produced with conventional graphics.

WHAT IS A SPRITE?

A sprite is an image composed of a number of small dots called **pixels**. Every character that is displayed on the screen is composed of a number of dots or pixels that are in the shape of the character. If a pixel is turned on, it shows as a bright spot on the screen. A pixel that is off is dark, or, in other words the screen color, so that it does not show. On a good quality screen, you can often see the dots in some letters such as X and Y.

A sprite is much the same except that you can define exactly where the dots should go that make up the sprite. While characters on the screen are created with a maximum of 64 dots, most use a lot fewer; sprites can use up to 504. The difference in the number of pixels means that sprites can be much larger than a character and yet can be moved around as simply as a single character graphic.

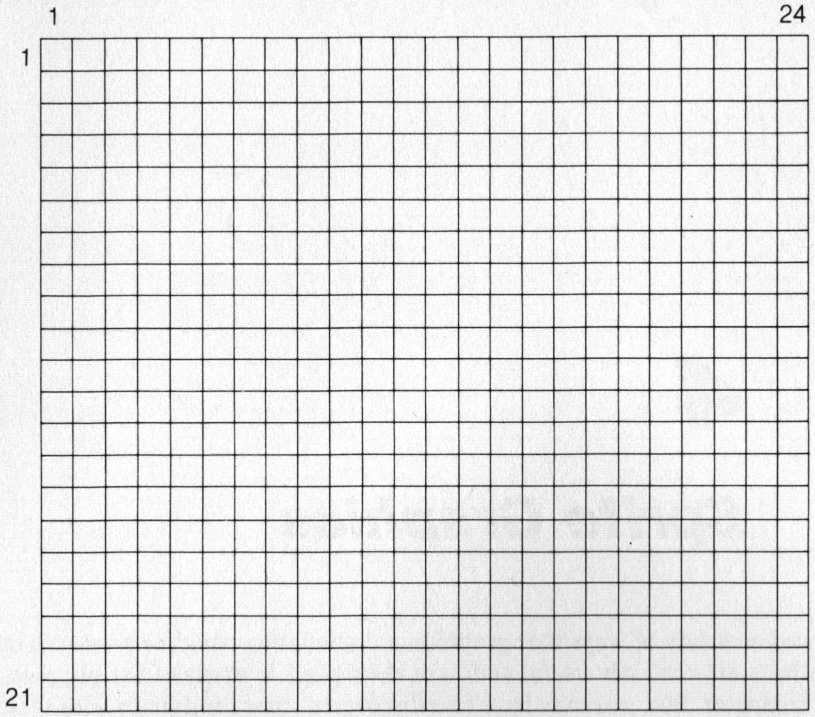


Figure 4.1 Grid for Sprite Design

SPRITE DESIGN

The 504 pixels used to create a sprite form a 24 × 21 grid as shown in figure 4.1. Using a grid like this one is quite helpful when designing your own sprites. Simple graph paper will do quite nicely if you mark off a rectangle 24 squares wide by 21 squares long. The 24 × 21 sprite will eventually be displayed on the screen, which itself is formed by a matrix of 320 by 200 dots.

The next step is to draw your sprite on this grid. Remember that each square represents a pixel and can be either on or off. Think of a pixel as a small light bulb. If you put a mark in a square then this means the light (pixel) will be turned on, otherwise it will be off.

Figure 4.2 shows a helicopter that has been designed on the sprite grid. Notice that each box in the grid is filled in completely, since you do not have the option of partially filling a pixel.

CONVERTING THE SPRITE TO NUMBERS

When your sprite has been designed, it will be necessary to translate the diagram into numbers that can be read by the program. The sprite grid is divided into three sections, each consisting of eight columns. The eight columns are given position values of 128, 64, 32, 16, 8, 4, 2, and 1 from left to right. Since there are three sections, there will be a total of 3 numbers times 21 rows or 63 numbers to represent the sprite.

To convert our chopper drawing into numbers refer to figure 4.3 for the values. Notice that the top five lines do not have any pixels marked so they are all zero values.

0,	0,	0
0,	0,	0
0,	0,	0
0,	0,	0
0,	0,	0

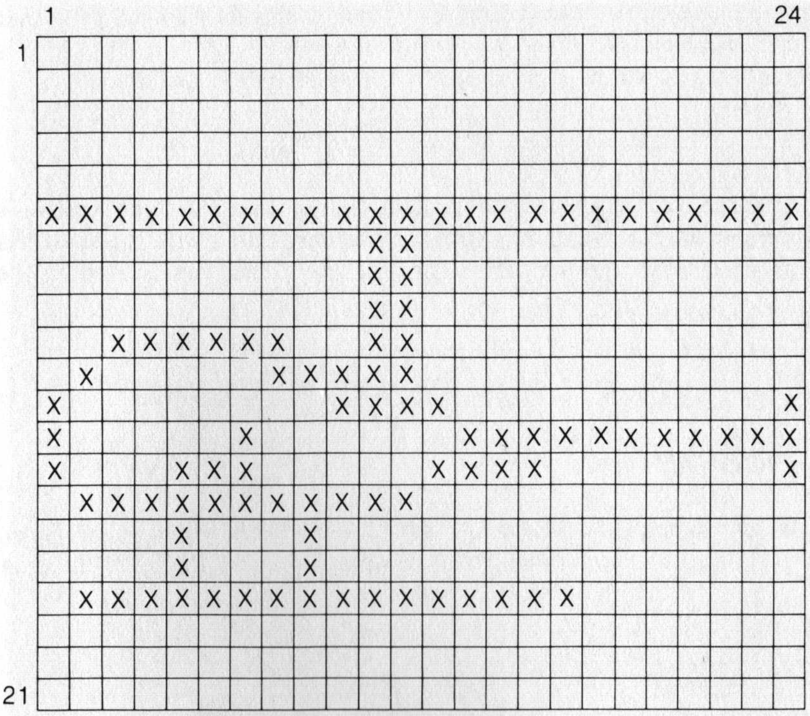


Figure 4.2 Sprite Design for a Helicopter

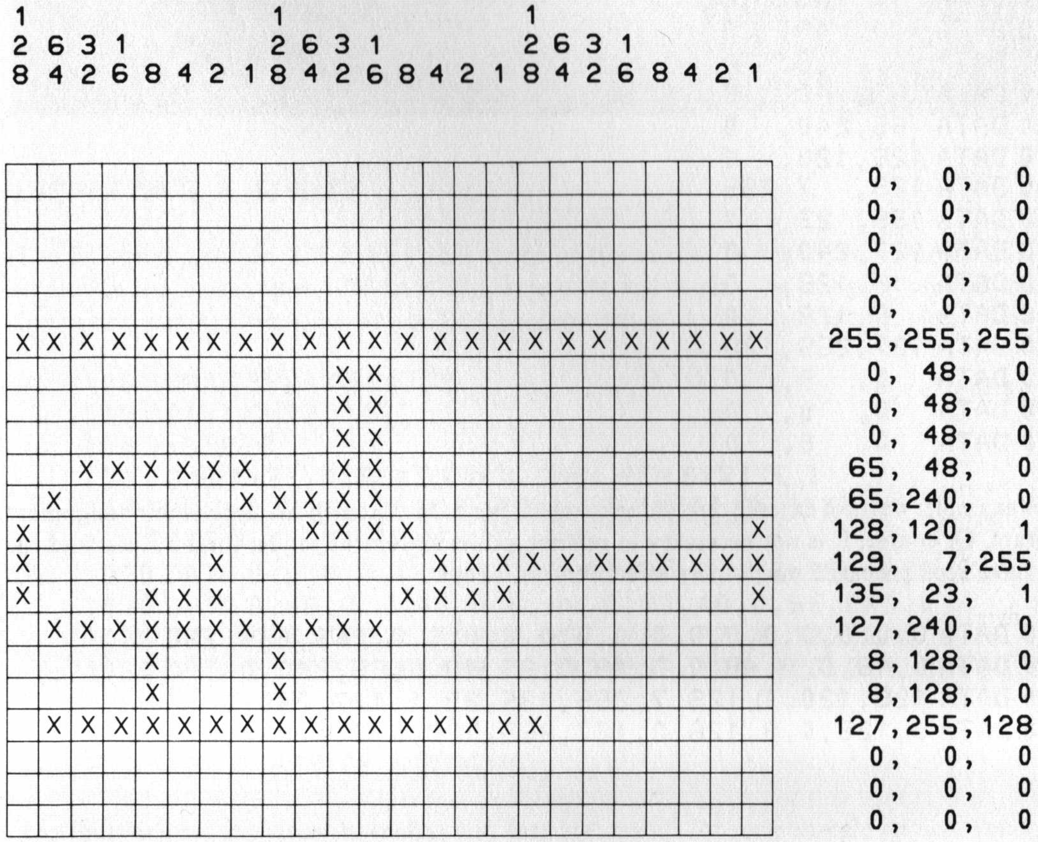

1 1 1
2 6 3 1 2 6 3 1 2 6 3 1
8 4 2 6 8 4 2 1 8 4 2 6 8 4 2 1 8 4 2 6 8 4 2 1
0, 0, 0
0, 0, 0
0, 0, 0
0, 0, 0
0, 0, 0
255,255,255
0, 48, 0
0, 48, 0
0, 48, 0
65, 48, 0
65,240, 0
128,120, 1
129, 7,255
135, 23, 1
127,240, 0
8,128, 0
8,128, 0
127,255,128
0, 0, 0
0, 0, 0
0, 0, 0

Figure 4.3 Assigning Numbers to the Sprite

The first line of the chopper represents the rotor blade. Every box in this line is filled in so that means each section will be the sum of the numbers 1 + 2 + 4 + 8 + 16 + 32 + 64 + 128 which is 255. In this case, each of the three sections will be represented by the numbers 255 as follows:

255, 255, 255

On the next line is a section of the rotor shaft. In this case only the center of the three sections is occupied, so the left and right sections are represented by zero. The center section with the shaft has the 16 and 32 columns drawn in which gives a value of 48. This full line is then represented by the values

0, 48, 0

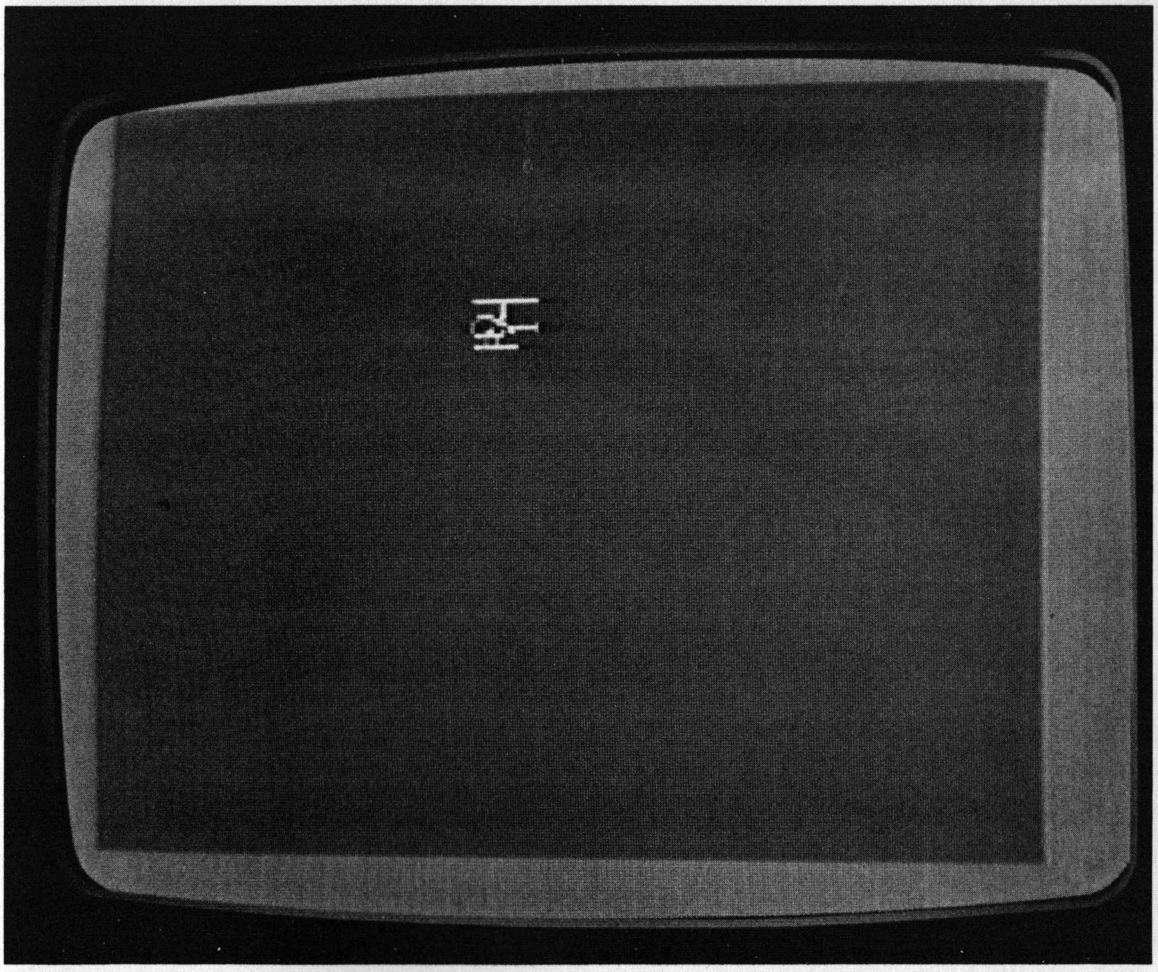
SPRITE DATA IN THE PROGRAM

The data used for the sprite is usually supplied in the form of data statements in a BASIC program. Taking the numbers allocated to our sprite drawing in figure 4.3, we can prepare the following data statements.

```
1000 DATA   0,  0,  0
1010 DATA   0,  0,  0
1020 DATA   0,  0,  0
1030 DATA   0,  0,  0
1040 DATA   0,  0,  0
1050 DATA 255,255,255
1060 DATA   0, 48,  0
1070 DATA   0, 48,  0
1080 DATA   0, 48,  0
1090 DATA  65, 48,  0
1100 DATA  65,240,  0
1110 DATA 128,120,  1
1120 DATA 129,  7,255
1130 DATA 135, 23,  1
1140 DATA 127,240,  0
1150 DATA   8,128,  0
1160 DATA   8,128,  0
1170 DATA 127,255,128
1180 DATA   0,  0,  0
1190 DATA   0,  0,  0
1200 DATA   0,  0,  0
```

In this example, we have entered only three numbers per data statement to match their assignment to the diagram. Of course, it is not necessary to restrict a data statement to just three values, and the data could have been prepared much more effectively as follows:

```
1000 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,255,255,255
1010 DATA 0,48,0,0,48,0,0,48,0,65,48,0,65,240,0
1020 DATA 128,120,1,129,7,255,135,23,1,127,240,0
1030 DATA 8,128,0,8,128,0,127,255,128,0,0,0,0,0,0,0,0,0
```



DISPLAYING A SPRITE

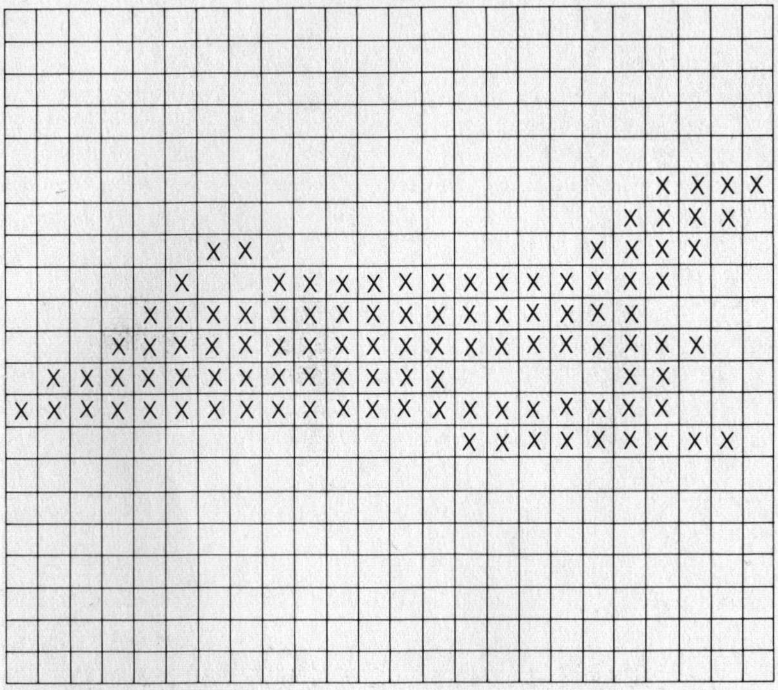
The following program will display your sprite on the screen. To use it just enter the data statements to replace the ones in lines 1000 to 1030. These lines will display the chopper designed in this chapter. In the next chapter, we will examine how this program works and how to animate sprites.

```
100 REM DISPLAY SPRITE
110 PRINT CHR$(147)
120 V=53248
130 POKE V+21,4:POKE 2042,195: REM SPRITE 3
140 FOR I=0 TO 62:READ D:POKE 195*64+I,D:NEXT
150 POKE V+4,150:POKE V+5,120
1000 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,255,255,255
1010 DATA 0,48,0,0,48,0,0,48,0,65,48,0,65,240,0
1020 DATA 128,120,1,129,7,255,135,23,1,127,240,0
1030 DATA 8,128,0,8,128,0,127,255,128,0,0,0,0,0,0,0,0,0
```

SOME SAMPLE SPRITE DESIGNS

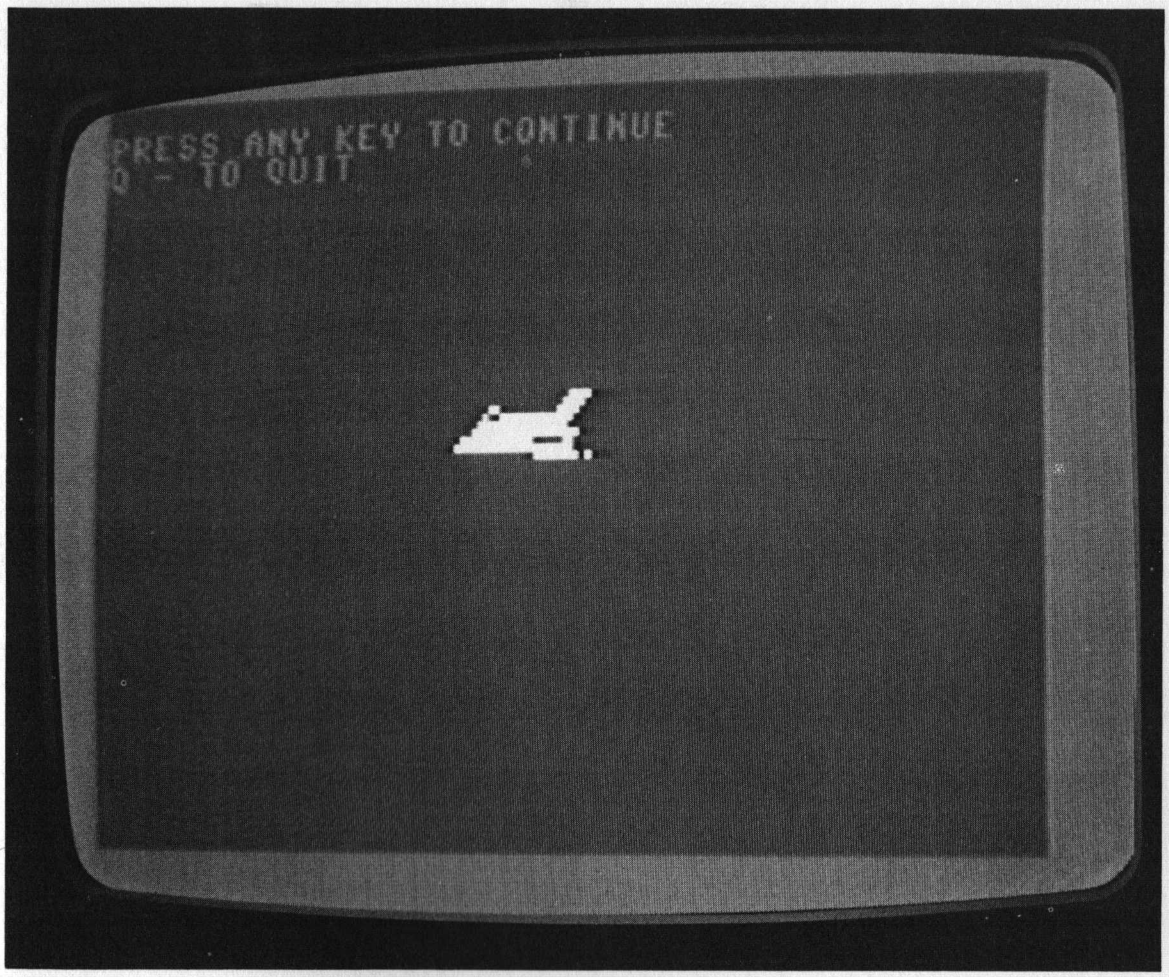
The following section gives the matrix and data statements for some sample sprites. You can use these in the above program or in a program of your own design to display or even animate the sprite.

```
1               1               1
2 6 3 1         2 6 3 1         2 6 3 1
8 4 2 6 8 4 2 1 8 4 2 6 8 4 2 1 8 4 2 6 8 4 2 1
```

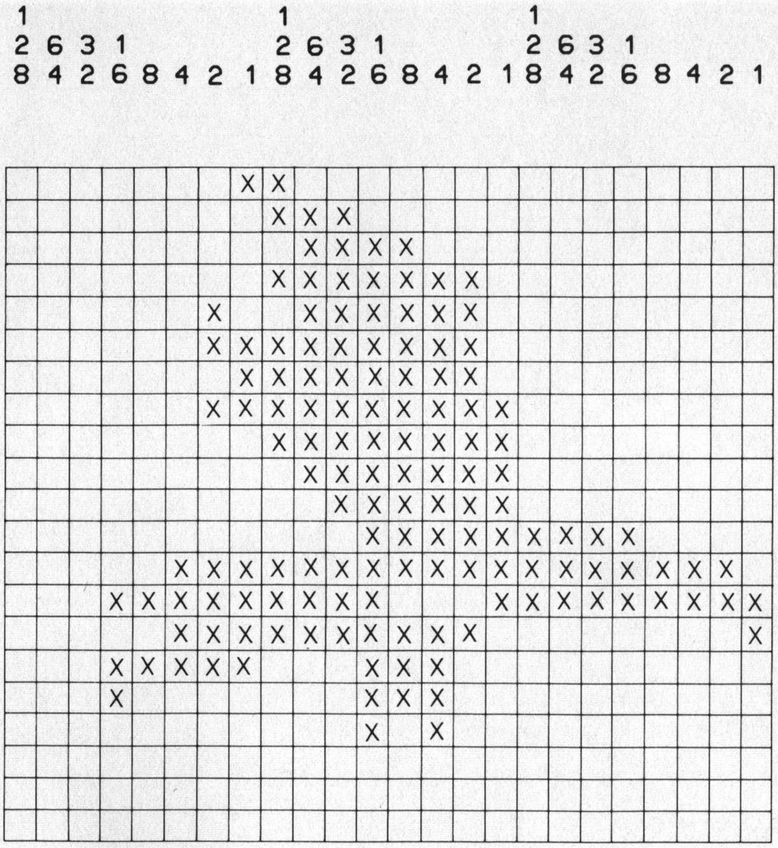


```
  0,  0,  0
  0,  0,  0
  0,  0,  0
  0,  0,  0
  0,  0,  0
  0,  0, 15
  0,  0, 30
  3,  0, 60
  4,255,248
 15,255,240
 31,255,252
127,252, 24
255,255,248
  0,  3,253
  0,  0,  0
  0,  0,  0
  0,  0,  0
  0,  0,  0
  0,  0,  0
  0,  0,  0
  0,  0,  0
```

```
1000 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,15

1010 DATA 0,0,30,3,0,60,4,255,248,15,255,240,31,255,252

1020 DATA 127,252,24,255,255,248,0,3,253,0,0,0

1030 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
```

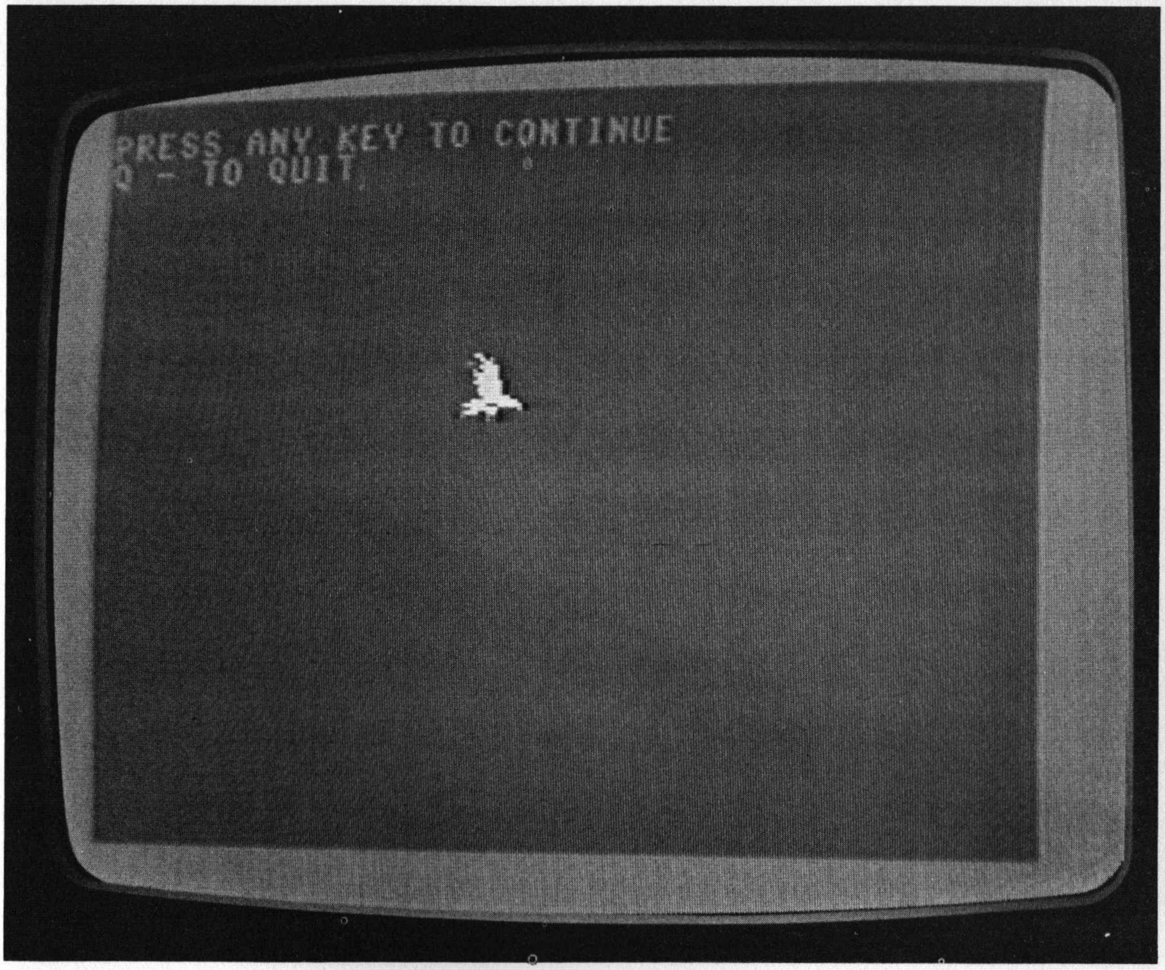


```
PRESS ANY KEY TO CONTINUE
Q - TO QUIT
```


128 64 32 16 8 4 2 1 128 64 32 16 8 4 2 1 128 64 32 16 8 4 2 1

```
  1,192,  0
  0,108,  0
  0, 60,  0
  0,127,  0
  1, 63,  0
  1,255,  0
  0,255,  0
  1,255,128
  0,127,128
  0,255,128
  0,127,128
  0, 63,240
  3,255,254
 31,240,255
  3,255,  1
 31,142,  0
 16, 10,  0
  0, 10,  0
  0,  0,  0
  0,  0,  0
  0,  0,  0
```

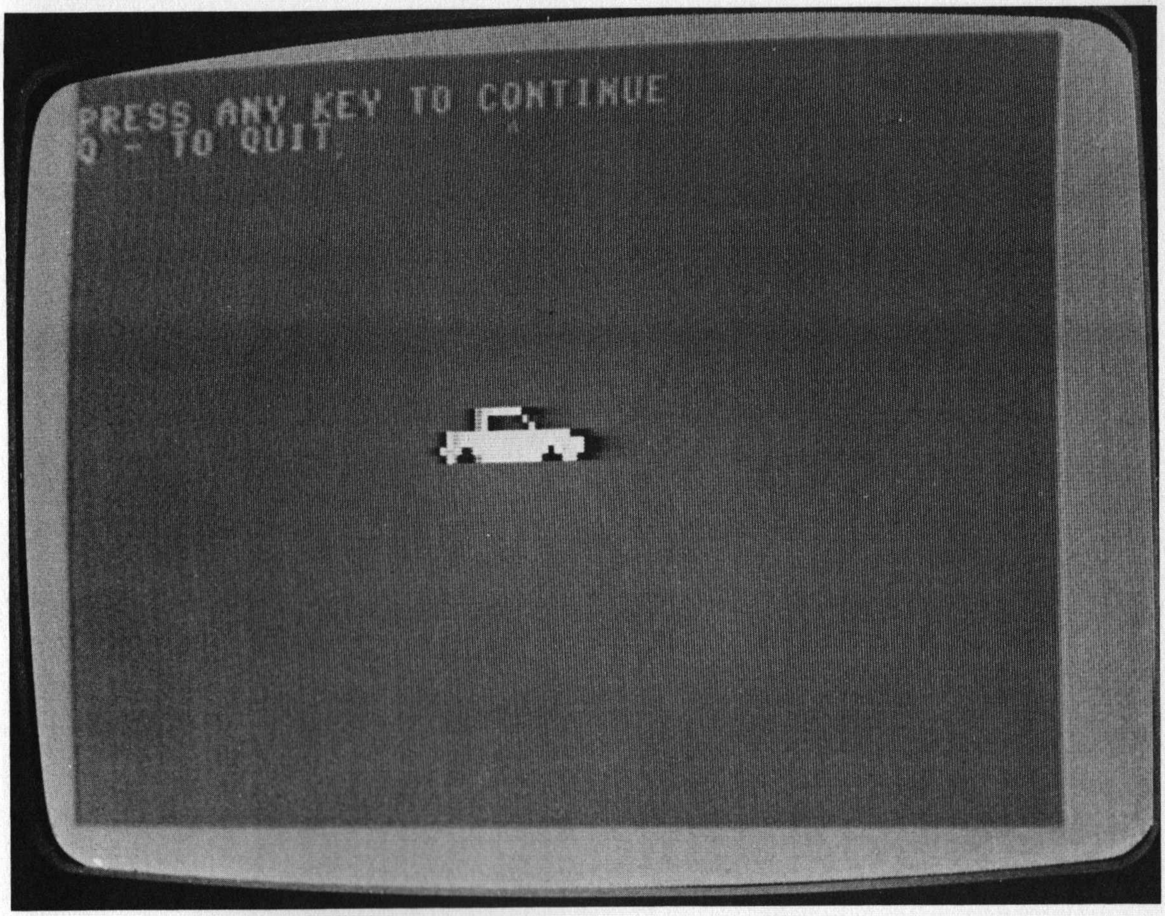
```
1000 DATA 1,192,0,0,108,0,10,60,0,0,127,0,1,63,0,1,255,0
1010 DATA 0,255,0,1,255,128,0,127,128,0,255,128,0,127,128,0,63,240
1020 DATA 3,255,254,31,240,255,3,255,1,31,142,0
1030 DATA 16,10,0,0,10,0,0,0,0,0,0,0,0,0,0
```



```
PRESS ANY KEY TO CONTINUE
Q - TO QUIT
```

128	64	32	16	8	4	2	1	128	64	32	16	8	4	2	1	128	64	32	16	8	4	2	1	
																								0, 0, 0
																								0, 0, 0
																								0, 0, 0
																								0, 0, 0
																								0, 0, 0
																								0, 0, 0
																								0, 0, 0
																								0, 0, 0
																								0, 0, 0
																								0, 0, 0
																								0, 0, 0
						X	X	X	X	X	X	X	X											3,252, 0
						X	X							X										3, 2, 0
						X	X								X									3, 1, 0
	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X			127,255,252
	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	127,255,255
X	X	X	X		X	X	X	X	X	X	X	X	X	X	X	X	X		X	X	X	X	X	247,255,223
	X	X				X	X	X	X	X	X	X	X	X	X	X				X	X	X		99,255,142
																								0, 0, 0
																								0, 0, 0
																								0, 0, 0

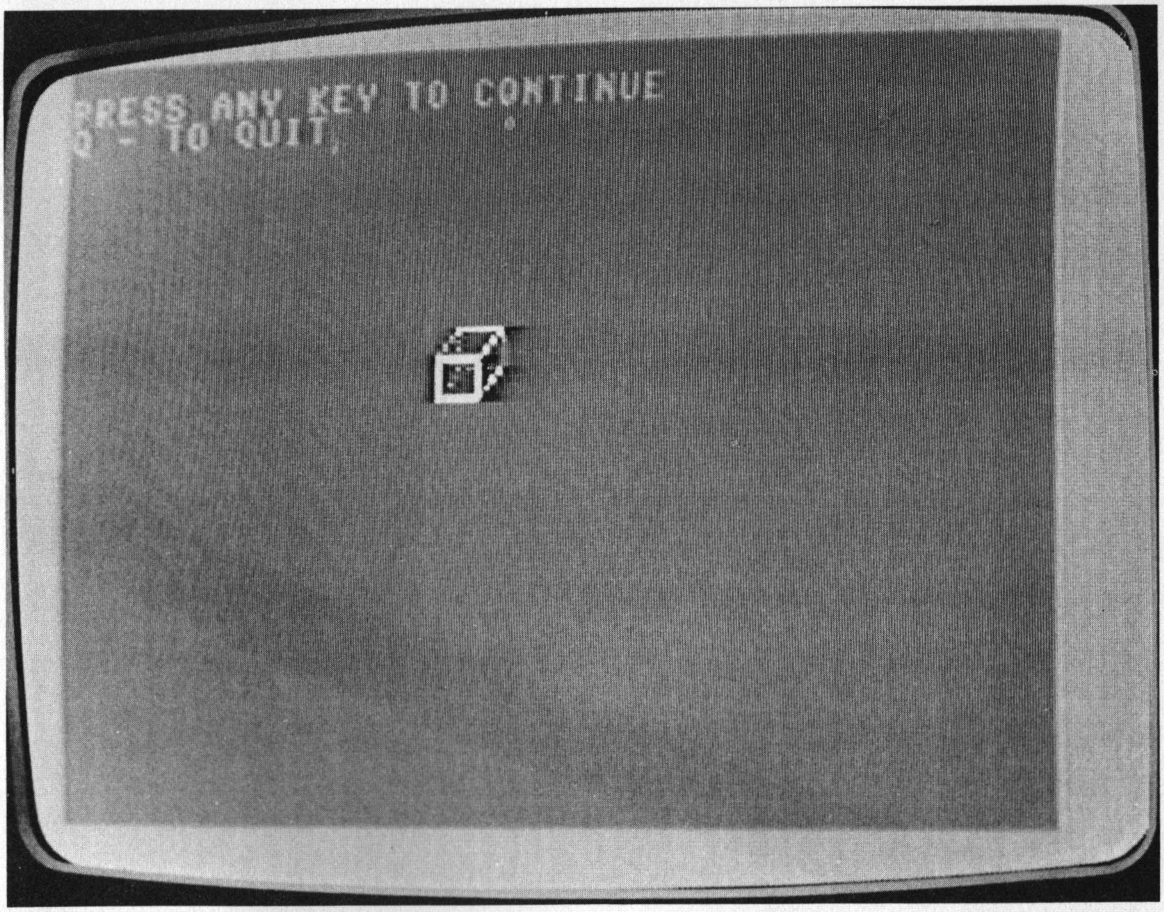
```
1000 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
1010 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,3,252,0
1020 DATA 3,2,0,3,1,0,127,255,252,127,255,255
1030 DATA 247,255,223,99,255,142,0,0,0,0,0,0,0,0,0
```



```
PRESS ANY KEY TO CONTINUE
Q - TO QUIT
```

128	64	32	16	8	4	2	1	128	64	32	16	8	4	2	1	128	64	32	16	8	4	2	1	
																								0, 0, 0
																								0, 0, 0
							X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		1,255,254
						X	X													X	X	X		3, 0, 14
					X														X	X		X		4, 0, 26
				X			X											X	X			X		9, 0, 50
			X														X	X				X		16, 0, 98
		X					X									X	X					X		33, 0,194
	X														X	X						X		64, 1,130
X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X							X		255,255, 2
X	X	X	X	X	X	X	X	X	X	X	X	X	X	X								X		255,254, 2
X	X												X	X								X		192, 6, 2
X	X						X		X		X		X	X		X		X			X	X		193, 86,166
X	X												X	X						X	X			192, 6, 12
X	X					X							X	X					X	X				194, 6, 24
X	X												X	X				X	X					192, 6, 48
X	X			X									X	X			X	X						200, 6, 96
X	X												X	X		X	X							192, 6,192
X	X	X											X	X	X	X								224, 7,128
X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X									255,255, 0
X	X	X	X	X	X	X	X	X	X	X	X	X	X	X										255,254, 0

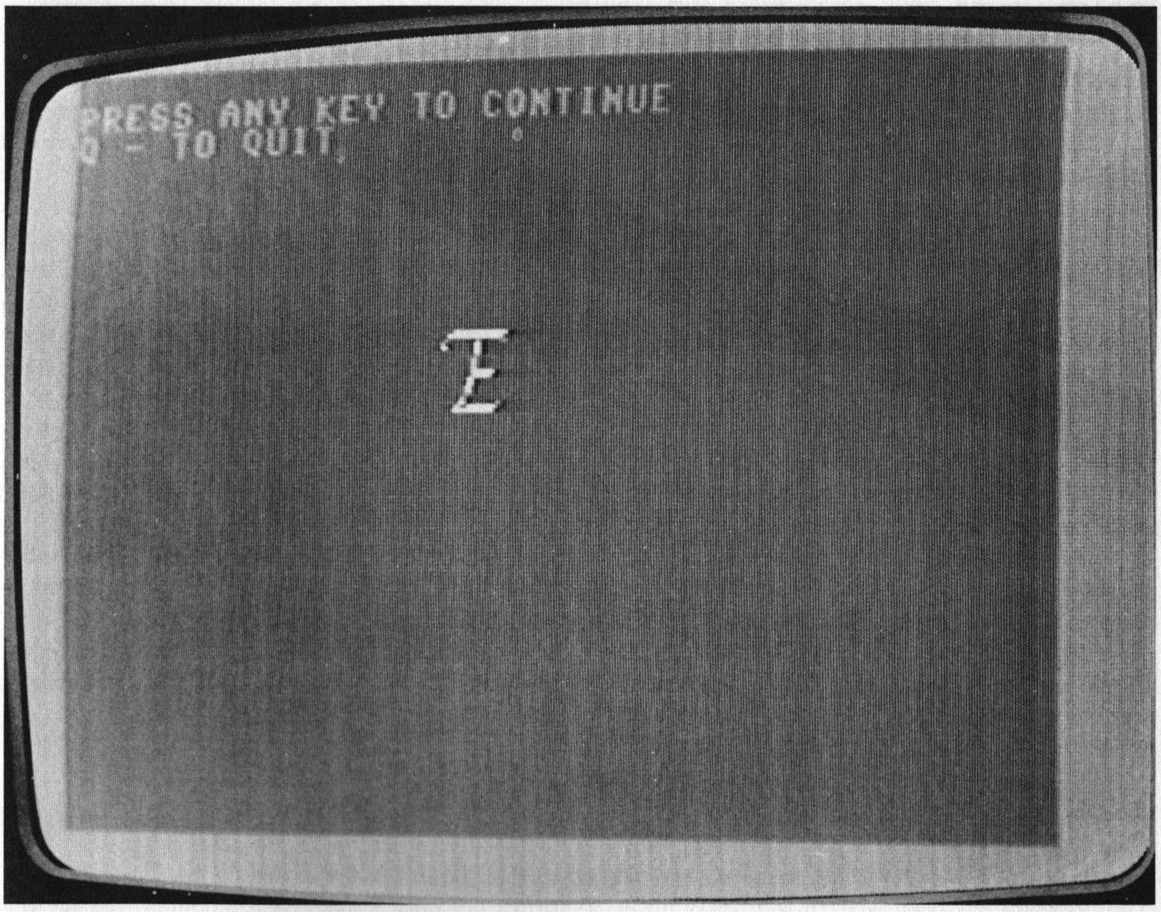
```
1000 DATA 0,0,0,0,0,0,1,255,254,3,0,14,4,0,26,9,0,50

1010 DATA 16,0,98,33,0,194,64,1,130,255,255,2,255,254,2

1020 DATA 192,6,2,193,86,166,192,6,12,194,6,24

1030 DATA 192,6,48,200,6,96,192,6,192,224,7,128

1040 DATA 255,255,0,255,254,0
```



```
PRESS ANY KEY TO CONTINUE
Q - TO QUIT,
```

128	64	32	16	8	4	2	1	128	64	32	16	8	4	2	1	128	64	32	16	8	4	2	1	
				X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X			15,255,252
		X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X				63,255,248
	X											X								X				64, 8, 8
X											X	X												128, 24, 0
X	X										X	X												192, 24, 0
											X	X												0, 24, 0
											X													0, 16, 0
										X	X													0, 48, 0
										X	X													0, 48, 0
										X	X													0, 48, 0
										X	X	X	X	X	X	X	X							0, 63,192
										X	X	X	X	X	X	X								0, 63,128
										X														0, 96, 0
									X	X														0, 96, 0
								X	X															0,192, 0
								X	X															0,192, 0
								X																0,128, 0
							X																	1, 0, 0
						X	X												X					3, 0, 16
					X	X			X	X	X	X	X	X	X	X	X	X						6,127,224
				X	X	X	X	X	X	X	X	X	X	X	X	X	X							15,255,192

```
100 DATA 15,255,252,63,255,248,64,8,8,128,24,0,192,24,0,0,24,0
110 DATA 0,16,0,0,48,0,0,48,0,0,48,0,0,63,192,0,63,128
120 DATA 0,96,0,0,96,0,0,192,0,0,192,0,0,128,0
130 DATA 1,0,0,3,0,16,6,127,224,15,255,192
```



```
PRESS ANY KEY TO CONTINUE
Q - TO QUIT.
```

SOME HINTS ON CREATING SPRITES

The process of creating a sprite is not all that difficult, but occasionally the results may not be exactly as you expected. One reason can be the height-to-width ratio. The height of a sprite is slightly greater than its width. Although this will not usually be a problem, there may be times when the difference can have an effect on the image you are trying to create.

Another thing to look for is the use of lines that are only one pixel in width. Lines made up of a single row of pixels, especially vertical lines, will be barely visible. The quality of your TV or monitor will also have an effect. One solution is to simply use two adjacent rows of pixels to make a heavier line.

Of course, the single line that is barely visible can also be used to good effect. In the preceding examples, single lines were used for the windshield on the car and the bubble on the chopper. In these sprites, the thin lines give the almost invisible quality of the glass and yet show where the glass line is located.

The sprite of the three-dimensional box used heavy double lines for the parts of the box that are nearest the viewer, while the single lines of pixels were used for the back of the box. Parts of the box that are normally hidden used dotted lines, as would be done on a draftsman's drawing. Dotted lines were made by using alternating pixels.

Another problem can occur if the total number of values for the sprite is not exactly 63. Remember there are 21 lines, and each line requires three values. If your list is either short of this number, or exceeds it, the sprite can take on some rather strange characteristics.

Later, when we get into the use of colored sprites, the choice of color will also be an important factor. Some experimentation may be required to create colored sprites with the color and characteristics you desire.

SELECT SPRITE PROGRAM

The following program will let you select and display each of the sprites developed on the previous pages. The program has been written to display up to 10 sprites but only 5 have been included, so that you can develop your own and add them to the program.

To add your own sprites, a few simple changes are needed. First, statement 140 should be changed from 5 to the number of sprites that you will have. If you add just one sprite to the existing ones, change this number to 6.

Secondly, you will want to add the name(s) of your sprite(s) to the menu. These statements are in lines 500 to 710. Just type your sprite name into the list of sprites, and they will become part of the menu.

Finally, add your sprite data to the end of the program.

```
100 REM SELECT SPRITE
110 DIM SD(10,63)
120 V=53248
130 POKE 2042,195:REM SPRITE 2
140 FOR S=1 TO 5
150 FOR D=0 TO 62
160 READ SD(S,D)
170 NEXT:NEXT
180 GOSUB 500
190 GOTO 180
500 REM SELECT SPRITE
510 PRINT CHR$(147);TAB(12);"SELECT SPRITE"
520 PRINT
530 PRINT TAB(8);" 1 - SPACE SHUTTLE"
```

```
540 PRINT
550 PRINT TAB(8);" 2 - EAGLE             "
560 PRINT
570 PRINT TAB(8);" 3 - CAR               "
580 PRINT
590 PRINT TAB(8);" 4 - 3-D BOX           "
600 PRINT
610 PRINT TAB(8);" 5 - LETTER            "
620 PRINT
630 PRINT TAB(8);" 6 -                   "
640 PRINT
650 PRINT TAB(8);" 7 -                   "
660 PRINT
670 PRINT TAB(8);" 8 -                   "
680 PRINT
690 PRINT TAB(8);" 9 -                   "
700 PRINT
710 PRINT TAB(8);"10 -                   "
715 PRINT
720 INPUT N
730 PRINT CHR$(147)
740 FOR I=0 TO 62:POKE 195*64+I,SD(N,I):NEXT
750 POKE V+4,150:POKE V+5,120
755 POKE V+21,4
760 PRINT "PRESS ANY KEY TO CONTINUE"
770 PRINT "Q - TO QUIT"
780 GET A$:IF A$="" THEN 780
790 IF A$="Q" THEN STOP
800 POKE V+21,0
810 RETURN
1000 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,15
1010 DATA 0,0,30,3,0,60,4,255,248,15,255,240,31,255,252
1020 DATA 127,252,24,255,255,248,0,3,253,0,0,0
1030 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
1040 DATA 1,192,0,0,108,0,0,60,0,0,127,0,1,63,0,1,255,0
1050 DATA 0,255,0,1,255,128,0,127,128,0,255,128,0,127,
    128,0,63,240
1060 DATA 3,255,254,31,240,255,3,255,1,31,142,0
1070 DATA 16,10,0,0,10,0,0,0,0,0,0,0,0,0,0
1080 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
1090 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,3,252,0
1100 DATA 3,2,0,3,1,0,127,255,252,127,255,255
1110 DATA 247,255,223,99,255,142,0,0,0,0,0,0,0,0,0
1120 DATA 0,0,0,0,0,0,1,255,254,3,0,14,4,0,26,9,0,50
1130 DATA 16,0,98,33,0,194,64,1,130,255,255,2,255,254,2
1140 DATA 192,6,2,193,86,166,192,6,12,194,6,24
1150 DATA 192,6,48,200,6,96,192,6,192,224,7,128
1160 DATA 255,255,0,255,254,0
1170 DATA 15,255,252,63,255,248,64,8,8,128,24,0,192,24,
    0,0,24,0
1180 DATA 0,16,0,0,48,0,0,48,0,0,48,0,0,63,192,0,63,128
1190 DATA 0,96,0,0,96,0,0,192,0,0,192,0,0,128,0
1200 DATA 1,0,0,3,0,16,6,127,224,15,255,192
```

Comments on the Program

Lines	
110	Array defined to hold data for up to 10 sprites. Since only 1 sprite is displayed at a time, it can be selected faster from the array than reading through all of the data each time.
110–130	Initialize registers.
140–170	Read the sprite data into array SD. The ending value of 5 in statement 150 can be changed to reflect additions to the program.
180	Go to the subroutine for selecting and displaying the sprite.
190	Repeats as often as desired.
500–810	Subroutine to select and display sprite.
510–715	List of sprites to select from.
720	Accepts user's choice.
730	Clears screen.
740	Loads sprite into memory.
750–755	Display the sprite. These pokes will be fully explained in the next chapter.
760–790	Option to continue or quit.
800	Erases sprite.
1000–1030	Data for the space shuttle.
1040–1070	Data for the eagle.
1080–1110	Data for the car.
1120–1160	Data for the three-dimensional box.
1170–1200	Data for the letter E.

Challenge

Prepare data for five of your own sprite designs and add them to the selection presented by this program.

5

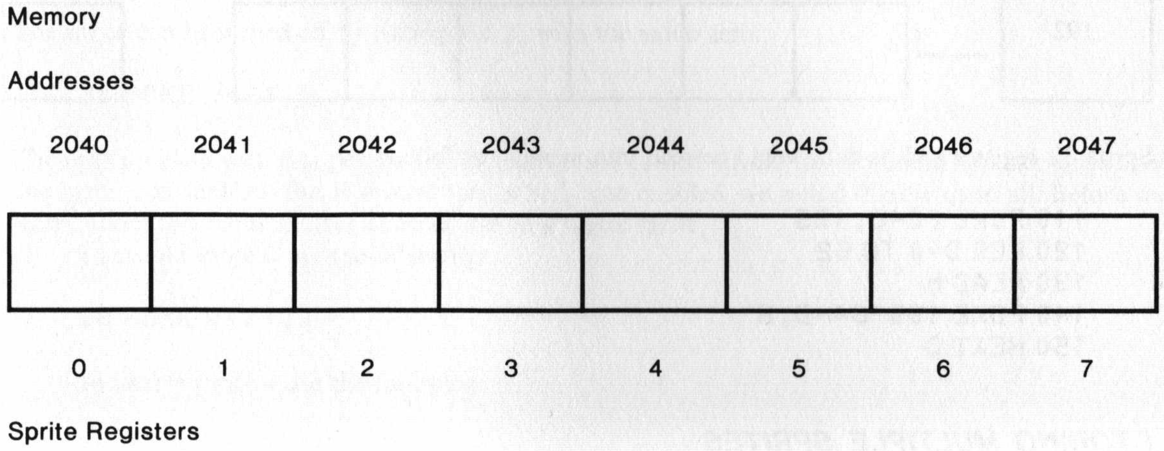
Sprite Graphics and Memory

To use sprites effectively, an understanding of the sprite's use of memory is essential. A maximum of eight sprites, numbered 0 to 7, may be used by a BASIC program. To help define these sprites, eight registers are used by the Commodore 64.

SPRITE REGISTERS

As we saw in the last chapter, every sprite is defined with 63 numeric values. To use a sprite, these values must first be poked into memory so that the video chip has access to the shape information that these numbers supply. Part of our task is to tell the computer where these numbers can be found; this is done through the sprite registers.

Sprite registers are located in memory addresses 2040 to 2047. As you can see in the diagram below, there are eight registers, one for each sprite. The register will contain a number that tells the computer where the 63 numbers for the sprite shape are stored. Since there are eight registers, we can store a maximum of eight different sprites in the program at one time.



If you are using sprite 0, then register 0 will be poked with the address of the data for sprite 0. The sprite program at the end of the last chapter used sprite 2, so address 2042 was poked with the location of sprite 2's data.

Table 5.1 Memory Allocation for Sprite Data

Sprite	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Register	2040	2041	2042	2043	2044	2045	2046	2047
Memory Area	192	193	194	195	196	197	198	199
Starting Poke Address for Data	192*64	193*64	194*64	195*64	196*64	197*64	198*64	199*64

AVAILABLE MEMORY FOR SPRITES

Any memory that is not being used for your program can be used for storing the sprite data. Table 5.1 shows memory locations that will be useful for most of your sprite applications. These memory locations are above where a BASIC program of reasonable size will be located.

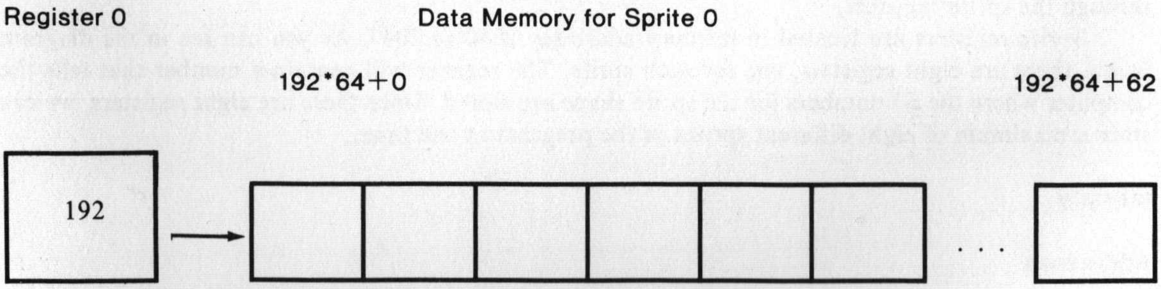
If your program is using sprite 0, then register 0 (at address 2040) will be poked with the pointer value 192 as follows:

```
110 POKE 2040,192
```

Data for sprite 0 would be read into addresses 192*64 to 192*64+62 for a total of 63 values representing the sprite's shape. A code such as the following can be used.

```
120 FOR D=0 TO 62
130 READ N
140 POKE 192*64+D,N
150 NEXT D
```

The following diagram shows how the Commodore 64 views this data organization:



Sprite 3's register and its data would be read into memory as follows:

```
110 POKE 2040,195
120 FOR D=0 TO 62
130 READ N
140 POKE 195*64+D,N
150 NEXT D
```

STORING MULTIPLE SPRITES

In some programs you will want to store more than one sprite in memory. In some cases you may even want to have all eight sprites defined. Although this will require a lot of data reading and storing, the sprite shapes will not be too difficult.

First we need to set the registers to locations 192 to 199 and then read and poke the sprite data values. In the following code, the variable NS refers to the number of sprites. In this case it is set to 7, but it could have any value from 0 to 7 depending on the sprites used.

```
100 NS=7
110 FOR R=0 TO NS:           REM NUMBER OF SPRITES AND REGISTERS
120 POKE 2040+R,192+R:       REM SET REGISTER
130 FOR S=0 TO 62:           REM SPRITE DATA LOOP
140 READ N:                  REM READ SPRITE DATA
150 POKE (192+R)*64+S,N:     REM POKE SPRITE DATA
160 NEXT S
170 NEXT R
```

ENABLING AND DISABLING SPRITES

This section might be better known as turning sprites on or off which is what enabling and disabling mean. Before a sprite can appear on the screen, it must be turned on; it must also be given coordinates, but we will look at that subject later.

The Commodore 64 has a special video processor chip called the VIC-II. It is this chip that controls the display of sprites on the screen. In the VIC-II chip are some registers that, among other things, control the enabling and disabling of the sprites. This chip is located at address 53248. Table 5.2 indicates that register 21 is used to enable and disable the sprites when the processor evaluates the value of each bit in the register.

ENABLING A SINGLE SPRITE

As table 5.2 shows, a sprite is enabled by setting the appropriate bit of the register to the value 1. It is disabled by setting the bit to 0. In the table, sprite 2 is enabled because the bit identifying sprite 2 has been set to the value 1. This bit can be set by poking V+21 with the decimal value 4 as the diagram indicates.

```
10 V=53248
20 POKE V+21,4
```

The sprite can be turned off by poking V+21 with the value zero.

```
20 POKE V+21,0
```

The only problem with this poke is that all sprites are turned off. So far this doesn't matter since only one sprite was enabled. But if several sprites had been enabled, we would disable them all. Before we worry about this problem, let's concentrate on a single sprite.

To enable sprite 0 use the following:

```
20 POKE V+21,1
```

To enable sprite 4 use the following:

```
20 POKE V+21,16
```

Table 5.2 Sprite Enable/Disable Register

V=53248

Sprite Enable Register V+21

7	6	5	4	3	2	1	0	Sprite number
0	0	0	0	0	1	0	0	0—disable 1—enable
128	64	32	16	8	4	2	1	Decimal value

DISABLING SPRITES

To disable all sprites simply set the enable/disable register to 0. This may be done easily with the following:

```
20 POKE V+21,0
```

Remember this statement will disable all of the sprites that are currently in use. This will be useful when you want to completely clear the screen before proceeding with some other activity. Clearing all sprites from the screen may be a good idea before starting new action, to ensure no stray sprites are still there from some previous action.

ENABLING SEVERAL SPRITES

There are two ways several sprites can be enabled. One method is to simply poke V+21 with a value that represents all of the sprites that are to be enabled. The second method will let you selectively enable a sprite without affecting other sprites that may be currently enabled.

Enabling Multiple Sprites

Suppose you want to enable sprites 0, 2, and 5. The chart of values in table 5.2 shows these sprites to be represented by the numbers 1, 4, and 32. The total of these numbers is 37. So to enable sprites 0, 2, and 5 just poke register V+21 with the value of 37.

```
20 POKE V+21,37
```

Using this poke will enable the three sprites we wanted. However, if other sprites had been enabled before this poke was executed, they will now be disabled.

Enabling Selective Sprites

What happens if you want to enable sprite 3 but don't want other active sprites to become disabled. If register V+21 were poked with the value of 2, sprite 3 would be enabled but all other sprites would be disabled.

The solution to this problem is to use a peek to look at register V+21's current value. Then OR the value of the new sprite against the register. Here is how sprite 3 (represented by value 2) would be selectively turned on.

```
20 POKE V+21,PEEK(V+21) OR 2
```

This statement will enable sprite 3 without affecting the current status of the other sprites.

To selectively enable sprite 6 use the following poke:

```
20 POKE V+21,PEEK(V+21) OR 64
```

Several sprites may be selectively enabled with one statement using this technique. For example, the following statement will enable sprites 3, 4, and 5 without affecting the others:

```
20 POKE V+21,PEEK(V+21) OR 56
```

DISABLING SELECTIVE SPRITES

We have seen how to disable all eight sprites by the following command:

```
20 POKE V+21,0
```

But there will be times when you want to disable selective sprites without disabling others. As you might have guessed, selective disabling can be done in a manner similar to selective enabling. The difference is in the use of the AND operation.

To disable sprite 4 (value 16), the following statement is used:

```
20 POKE V+21,PEEK(V+21) AND (255-16)
```

Sprites 3 and 5 could be disabled without affecting sprite 4 or any other enabled sprites by using the following statement:

```
20 POKE V+21,PEEK(V+21) AND (255-40)
```

UNDERSTANDING BOOLEAN AND'S AND OR'S

When enabling and disabling sprites, we often use the Boolean operators AND to disable and OR to enable. These operators have the following properties when applied to the binary numbers 0 and 1.

AND Operation	*OR Operation*
0 AND 0 = 0	0 OR 0 = 0
0 AND 1 = 0	0 OR 1 = 1
1 AND 0 = 0	1 OR 0 = 1
1 AND 1 = 1	1 OR 1 = 1

OR to Enable a Sprite

Sprites are enabled by setting the bit to 1 which represents the sprite in register V+21. If sprites 2 and 3 are enabled, register V+21 will appear, as follows, with the decimal value 12.

7	6	5	4	3	2	1	0	Sprite numbers 2 and 3 enabled
0	0	0	0	1	1	0	0	V+21
128	64	32	16	8	4	2	1	Decimal value

The OR is used as follows to turn on sprites 0 and 5 without disabling the other sprites. The number 33 is called a mask. Its binary value has ones in the position for sprites 0 and 5. The OR acts on each bit in V+21 and or's them against the bits in the mask.

```
100 POKE (V+21),PEEK(V+21) OR 33
```

7	6	5	4	3	2	1	0	Sprite numbers 2 and 3 enabled
0	0	0	0	1	1	0	0	PEEK(V+21)
128	64	32	16	8	4	2	1	Decimal value
0	0	1	0	0	0	0	1	Mask field OR 33 (Sprites 0 and 5)
0	0	1	0	1	1	0	1	After POKE(V+21)

AND to Disable a Sprite

Sprites may be disabled by using the Boolean AND operation. In the following example, the mask must contain ones for each sprite that is to remain enabled. The sprite that is to be disabled is represented by a zero in the mask. This can be achieved by subtracting the value of the sprite(s) to be disabled from 255. Note that 255 is 11111111 in binary notation. In this example, only sprite 2 is to be disabled.

```
100 POKE (V+21),PEEK(V+21) AND (255=4)
```

7	6	5	4	3	2	1	0	Sprite numbers 0, 2, 3 and 5 enabled
0	0	1	0	1	1	0	1	PEEK(V+21)
128	64	32	16	8	4	2	1	Decimal value
1	1	1	1	1	0	1	1	Mask field OR (255-4) (Sprite 2)
0	0	1	0	1	0	0	1	After POKE(V+21)

SPRITE MOVEMENT

Sprites are given movement on the screen by varying the values of the X and Y registers listed in table 5.3. Each sprite has two registers, one to define the position along the X axis and the other for the Y axis. Each register is poked with a value from 0 to 255 to position the sprite on the axis. Since the X axis has 320 visible bit positions on the screen, a second register, called the MSB, is used to identify movement beyond position 255.

Table 5.3 Sprite X and Y Coordinate Registers

Register	Sprite Coordinate
0	X Coordinate—Sprite 0
1	Y Coordinate—Sprite 0
2	X Coordinate—Sprite 1
3	Y Coordinate—Sprite 1
4	X Coordinate—Sprite 2
5	Y Coordinate—Sprite 2
6	X Coordinate—Sprite 3
7	Y Coordinate—Sprite 3
8	X Coordinate—Sprite 4
9	Y Coordinate—Sprite 4
10	X Coordinate—Sprite 5
11	Y Coordinate—Sprite 5
12	X Coordinate—Sprite 6
13	Y Coordinate—Sprite 6
14	X Coordinate—Sprite 7
15	Y Coordinate—Sprite 7
16	Most Significant Bit (MSB)

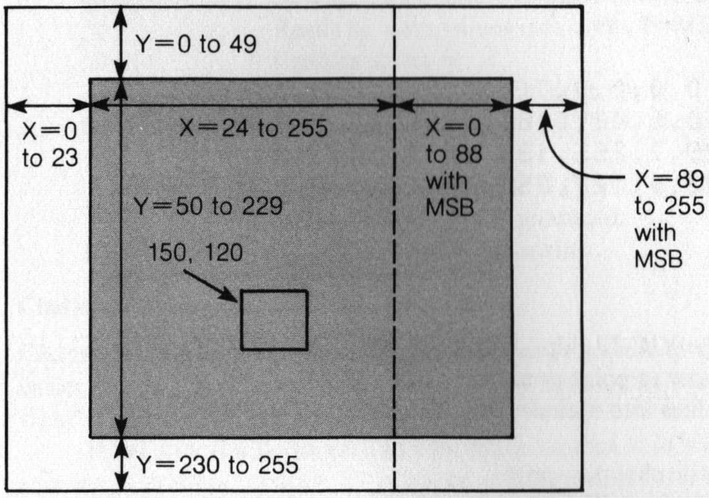


Figure 5.1 X and Y Screen Coordinates

SCREEN POSITIONING OF SPRITES

Sprites are positioned on the screen by poking the X and Y register for that sprite. Sprite 0 would be positioned by poking coordinate values into registers V+0 and V+1. The program to select and display a sprite in the preceding chapter used the statement:

```
750 POKE V+4,150:POKE V+5,120
```

These pokes were for sprite 2. Register V+4, the X coordinate, was set at position 150; and register V+5, the Y coordinate, was set at location 120. These coordinates refer to the position of the upper left corner of the sprite. If the sprite's image uses less than the full space available, the upper left corner of the space available for the sprite is the coordinate position.

Figure 5.1 shows the valid coordinates for the two dimensions. Notice that each border of the screen has an area where a sprite will be located off the visible viewing area of the screen. This area is useful for moving the sprite off the screen or on to it when using animation.

SPRITE ANIMATION

To animate a sprite, it is necessary to combine the steps outlined in the previous pages into a single program. In case all of this material has not quite come together yet, these steps are summarized as follows:

1. Set the sprite register (2040–2047) to point to the sprite data in memory.
2. Read sprite data into memory.
3. Enable the sprite (register V+21).
4. Vary the X and Y coordinates to give the sprite movement.

Now let's put all of this together into a program to show how sprite animation actually works. For this purpose the chopper sprite, from the last chapter, will be used. Try this program and see how the chopper appears in the sky and flies down to land in the lower left corner of the screen.

```
10 REM LAND CHOPPER
20 PRINT CHR$(147)
30 V=53248
40 POKE 2040,192
50 FOR D=0 TO 62:READ N:POKE 192*64+D,N:NEXT
60 POKE V+21,1
70 FOR Y=50 TO 230
80 POKE V+0,255-Y:POKE V+1,Y
90 NEXT Y
1000 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,255,255,255
1010 DATA 0,48,0,0,48,0,0,48,0,65,48,0,65,240,0
1020 DATA 128,120,1,129,7,255,135,23,1,127,240,0
1030 DATA 8,128,0,8,128,0,127,255,128,0,0,0,0,0,0,0,0,0
```

Comments on the Program

Lines	
20	Clears screen.
30	Sets address of the VIC-II chip.
40	Sets sprite 0 register to point to memory bank 192.
50	Reads 63 data values into memory for sprite 0.
60	Enables sprite 0.
70–90	Loop to animate the chopper sprite.
80	X coordinate for sprite 0 (V+0) is allocated values of 255–Y as Y varies from 50 to 230.
	Y coordinate for sprite 0 (V+1) is allocated the values in Y from 50 to 230.
1000–1030	Data for the chopper sprite.

Challenge

Change one or more of the coordinates, so that the chopper will fly off the screen when it reaches the bottom left corner.

MOVING A SPRITE HORIZONTALLY

A sprite may be moved horizontally across the screen using a technique similar to that applied in Chapter 3. In both cases the Y coordinate is held at a constant location, and the X coordinate is varied to provide movement in either a left to right or opposite direction.

In the following program, the chopper is animated by moving it from the right to the left side of the screen. In this example, the Y coordinate is held constant at position 100 and the X value is varied from 0 to 200. This number is then subtracted from 255 to produce movement from right to left across the screen.

```
10 REM CHOPPER R TO L
20 PRINT CHR$(147)
30 V=53248
40 POKE 2040,192
50 FOR D=0 TO 62:READ N:POKE 192*64+D,N:NEXT
60 POKE V+21,1
70 FOR X=0 TO 200
80 POKE V+0,255-X:POKE V+1,100
90 NEXT X
1000 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,255,255,255
1010 DATA 0,48,0,0,48,0,0,48,0,65,48,0,65,240,0
1020 DATA 128,120,1,129,7,255,135,23,1,127,240,0
1030 DATA 8,128,0,8,128,0,127,255,128,0,0,0,0,0,0,0,0
```

Comments on the Program

Lines	
20	Clears screen.
30	Sets address of the VIC-II chip.
40	Sets sprite 0 register to point to memory bank 192.
50	Reads 63 data values into memory for sprite 0.
60	Enables sprite 0.
70–90	Loop to animate the chopper sprite.
80	X coordinate for sprite 0 (V+0) is allocated values of 255–X as X varies from 0 to 200.
	Y coordinate for sprite 0 (V+1) is allocated the value of 100 so that no vertical movement is produced.
1000–1030	Data for the chopper sprite.

Challenge

Change the above program to fly the chopper from left to right across the screen. Since you got the easier end of the problem this time, did you notice that the chopper is flying backward from left to right? Looks weird, doesn't it?

What can you do to reverse the image so that it is flying forward but from left to right? One solution is to simply redraw the chopper and enter new data. But, can you find a way to use the current data and load it in such a way that the image is reversed?

MOVING A SPRITE VERTICALLY

Like horizontal movement, moving a sprite vertically is similar to animating screen graphics. The X coordinate is held constant while the Y coordinate is varied to produce movement from the top of the screen down or from the bottom to the top.

In the next program, the chopper is moved from the top of the screen to land at the bottom. The ending value 230 for the Y coordinate is chosen to position the chopper at the bottom of the screen, when it stops moving.

```
10 REM CHOPPER T TO B
20 PRINT CHR$(147)
30 V=53248
40 POKE 2040,192
50 FOR D=0 TO 62:READ N:POKE 192*64+D,N:NEXT
60 POKE V+21,1
```

```
70 FOR Y=50 TO 230
80 POKE V+0,150:POKE V+1,Y
90 NEXT Y
1000 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,255,255,255
1010 DATA 0,48,0,0,48,0,0,48,0,65,48,0,65,240,0
1020 DATA 128,120,1,129,7,255,135,23,1,127,240,0
1030 DATA 8,128,0,8,128,0,127,255,128,0,0,0,0,0,0,0,0,0
```

Comments on the Program

Lines	
10–60	Initialize registers and sprite memory.
70–90	Loop to animate the chopper sprite.
80	X coordinate for sprite 0 (V+0) is allocated a value of 150 to hold the sprite in a constant horizontal position. Y coordinate for sprite 0 (V+1) is allocated the values of 50 to 230 to produce vertical movement from the top of the screen to the bottom.
1000–1030	Data for the chopper sprite.

Challenge

Revise the preceding program to start the chopper at the bottom of the screen, take off into the sky, and disappear at the top of the screen.

SPEEDING UP YOUR SPRITES

The previous programs were useful for demonstrating the methods for animating a sprite. But they might have moved a bit slowly for your taste. What if you are writing a program for which fast action to add excitement to the play is important?

It is possible to move sprites much faster in BASIC. Quite simply, you just increment the loop controlling their movement by a value greater than one. For example, in the program that moved the chopper from right to left, the loop in statement 70 incremented X by a value of 1 each time. Therefore, it required 201 times through the loop to move the sprite from right to left.

What if we were to increment this loop by 10 each time? Then the program would only require 20 passes through the loop to move the sprite from right to left. Here is the revised program.

```
10 REM CHOPPER FAST
20 PRINT CHR$(147)
30 V=53248
40 POKE 2040, 192
50 FOR D=0 TO 62:READ N:POKE 192*64+D,N:NEXT
60 POKE V+21,1
70 FOR X=0 TO 200 STEP 10
80 POKE V+0,255-X:POKE V+1,100
90 NEXT X
1000 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,255,255,255
1010 DATA 0,48,0,0,48,0,0,48,0,65,48,0,65,240,0
1020 DATA 128,120,1,129,7,255,135,23,1,127,240,0
1030 DATA 8,128,0,8,128,0,127,255,128,0,0,0,0,0,0,0,0,0
```

Notice that the only change to the program is in line 70. It now reads

```
70 FOR X=0 TO 200 STEP 10
```

Run this program, and notice how much faster the chopper moves across the screen. Now make another change. This time use a step value of 2 as follows:

```
70 FOR X=0 TO 200 STEP 2
```

When the program is run with a step of 2, you will notice a slightly jerky motion in the movement. The reason for this is that we are moving the sprite 2 pixels at a time, and so it seems to jump to the next position, therefore reducing the smoothness of movement.

If the step value is increased to 3 and then 4 and so on, the sprite will move faster each time. As the speed increases, the jerky motion will tend to be reduced since the eye can't keep up with the quick motion of the sprite. Experiment with different step values and observe the different effects you can produce.

USING THE MSB REGISTER FOR FULL SCREEN WIDTH

As a sprite moves across the screen, the X register can be incremented to a maximum value of 255. This leaves the right-most 89 positions of the screen and the positions off to the right margin inaccessible to the sprite. This problem can be cured by using the Most Significant Bit (MSB) register located at V+16.

Most Significant Bit Register V+16

7	6	5	4	3	2	1	0	Sprite number
0	0	0	0	0	0	0	1	0—off 1—on
128	64	32	16	8	4	2	1	Decimal value

To use this register set the bit for the sprite to a one value when you want it to enter the MSB area of the screen. Then the X register will start at zero and increase to the new value.

In the previous example, the MSB register has been set for sprite 1. Individual sprites can be enabled and disabled in the MSB area by using the OR and AND methods discussed earlier in this chapter.

To move a sprite from left to right and have it move smoothly into the MSB area, program code like the following can be used.

```
100 POKE V+21,1:            REM ENABLE SPRITE
110 FOR X=0 TO 255:         REM MOVE THE FIRST 256 POSITIONS
120 POKE V+0,X:POKE V+1,100
130 NEXT X
140 POKE V+16,1:            REM SET MSB REGISTER
150 FOR X=0 TO 88:          REM MOVE INTO MSB AREA
160 POKE V+0,X:POKE V+1,100
170 NEXT X
```

This routine works well, but it does require a lot of statements. An alternative is to compute the X position and the poke value for the MSB as the program is looping between the real values of 0 to 343 along the X axis. This approach would result in the following code which makes use of the INT function.

```
100 POKE V+21,1:            REM ENABLE SPRITE
110 FOR X=0 TO 343          REM SET X AXIS
120 M=INT(X/256):           REM 0 OR 1 VALUE FOR MSB
130 POKE V+16,M:            REM POKE MSB
140 POKE V+0,X-(256*M):     REM SET X COORDINATE
150 POKE V+1,100:           REM SET Y COORDINATE
160 NEXT X
```

Although this code is more compact, it is more complex than the previous solution. It will also run slower, which may be a disadvantage in some programs. It does have the advantage of easily computing the position, at any time, of the X coordinate and the value of the MSB register. This proves useful if a sprite is moving in both the X and Y directions and frequently crosses the MSB boundary on the horizontal axis.

The following program uses the MSB to move the chopper from the right side of the screen and off the screen at the left margin.

```
10 REM CHOPPER MSB
20 PRINT CHR$(147)
30 V=53248
40 POKE 2040,192
50 FOR D=0 TO 62:READ N:POKE 192*64+D,N:NEXT
60 POKE V+21,1
70 FOR X=343 TO 0 STEP -1
80 M=INT(X/256)
90 POKE V+16,M
100 POKE V+0,X-(256*M):POKE V+1,100
110 NEXT X
1000 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,255,255,255
1010 DATA 0,48,0,0,48,0,0,48,0,65,48,0,65,240,0
1020 DATA 128,120,1,129,7,255,135,23,1,127,240,0
1030 DATA 8,128,0,8,128,0,127,255,128,0,0,0,0,0,0,0,0,0
```

Comments on the Program

Lines	
10–60	Initialize registers and sprite memory.
70–110	Loop to animate the chopper sprite.
80	M is 0 when X is between the values of 0 and 255. Therefore, the MSB part of the horizontal axis will not be in use. When X is 256 or greater, M will become 1, and the MSB will be activated to use the right part of the horizontal axis.
90	Pokes the MSB register with either 0 or 1.
100	When X is a value between 0 and 255, the X coordinate will result in the X value. When X exceeds 255, the formula calculates the position of the X coordinate within the MSB area.
1000–1030	Data for the chopper sprite.

Challenge

The program called LAND CHOPPER included earlier in the chapter takes the chopper from the top of the screen, near the MSB border, and moves it to the bottom left corner. Revise this program so that the sprite begins its landing pattern from the top right corner of the screen, which is located within the MSB area.

SPRITE COLOR

The color of each sprite can also be controlled by the program. This is done by setting a color register to the color code that you want. The color registers are listed in table 5.4. Colors are selected from values 0 to 15 for screen and border colors as described earlier in the book.

To set sprite 2 to yellow, poke register 41 with yellow's code which is 7, as follows:

```
100 POKE V+41,7
```

Table 5.4 Sprite Color Registers and Codes

Register	Sprite	Color Codes			
V+39	0	Black	0	Orange	8
V+40	1	White	1	Brown	9
V+41	2	Red	2	Light Red	10
V+42	3	Cyan	3	Dark Gray	11
V+43	4	Purple	4	Medium Gray	12
V+44	5	Green	5	Light Green	13
V+45	6	Blue	6	Light Blue	14
V+46	7	Yellow	7	Light Gray	15

Sprite 5 can be set to light green with the statement

```
100 POKE V+44,13
```

SPRITE SIZE

A sprite's size may be increased by expanding it in the X or Y direction, or both. This is achieved by using the expansion registers at V+23 and V+29 (see below). Register 23 is for expansion in the Y direction and 29 is used to expand a sprite in the X direction. The bits of each register are poked to identify the sprite or sprites to be expanded.

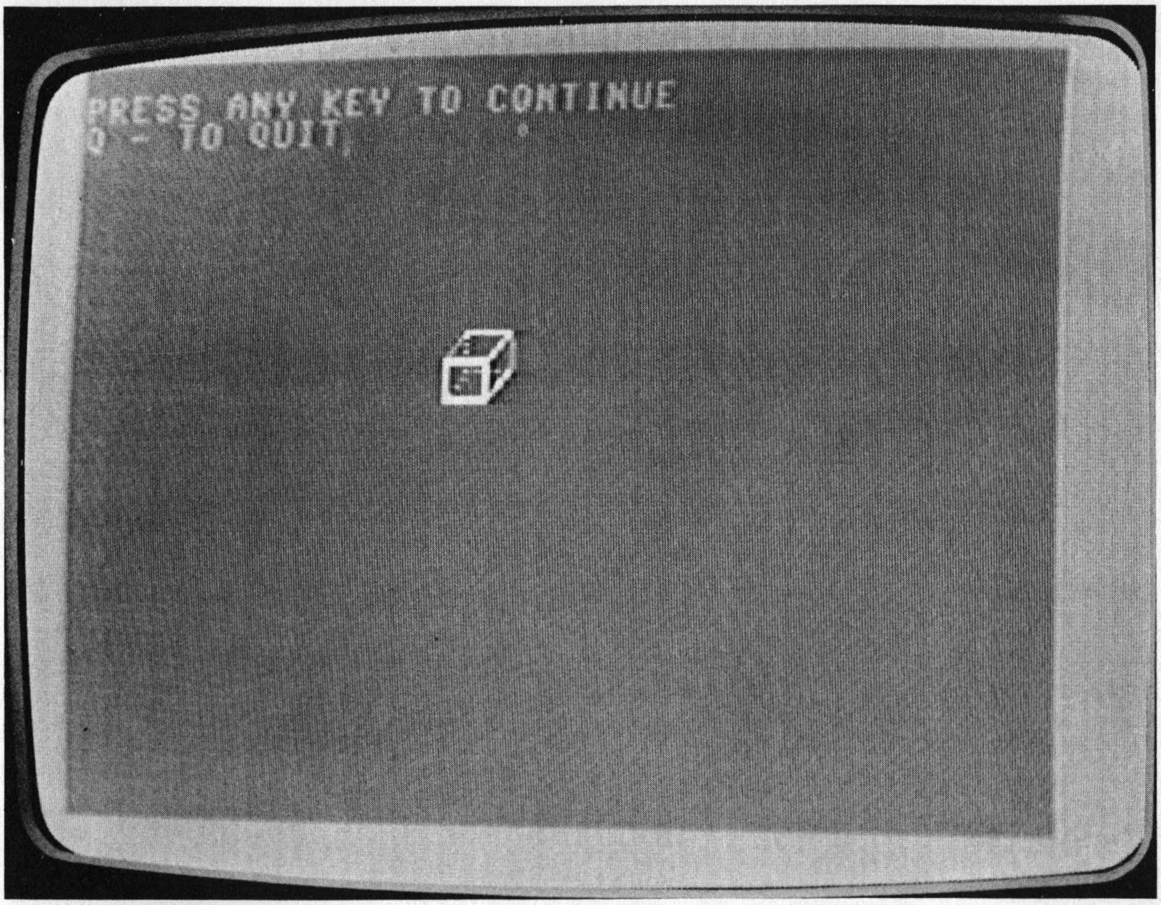
7	6	5	4	3	2	1	0	Sprite numbers 2 and 3 enabled
0	0	0	0	0	1	0	0	Expansion register V+23 Expand in Y direction
0	0	0	0	0	1	0	0	Expansion register V+29 Expand in X direction
128	64	32	16	8	4	2	1	Decimal value

To expand sprite 2 in both the X and Y axes, both registers will be poked with a 4, identifying sprite 2 for expansion as follows:

```
100 POKE V+23,4: POKE V+29,4
```

Expanding the sprite in the Y direction but not in the X would be accomplished with the following statement:

```
100 POKE V+23,4: POKE V+29,0
```



```
PRESS ANY KEY TO CONTINUE
Q - TO QUIT.
```

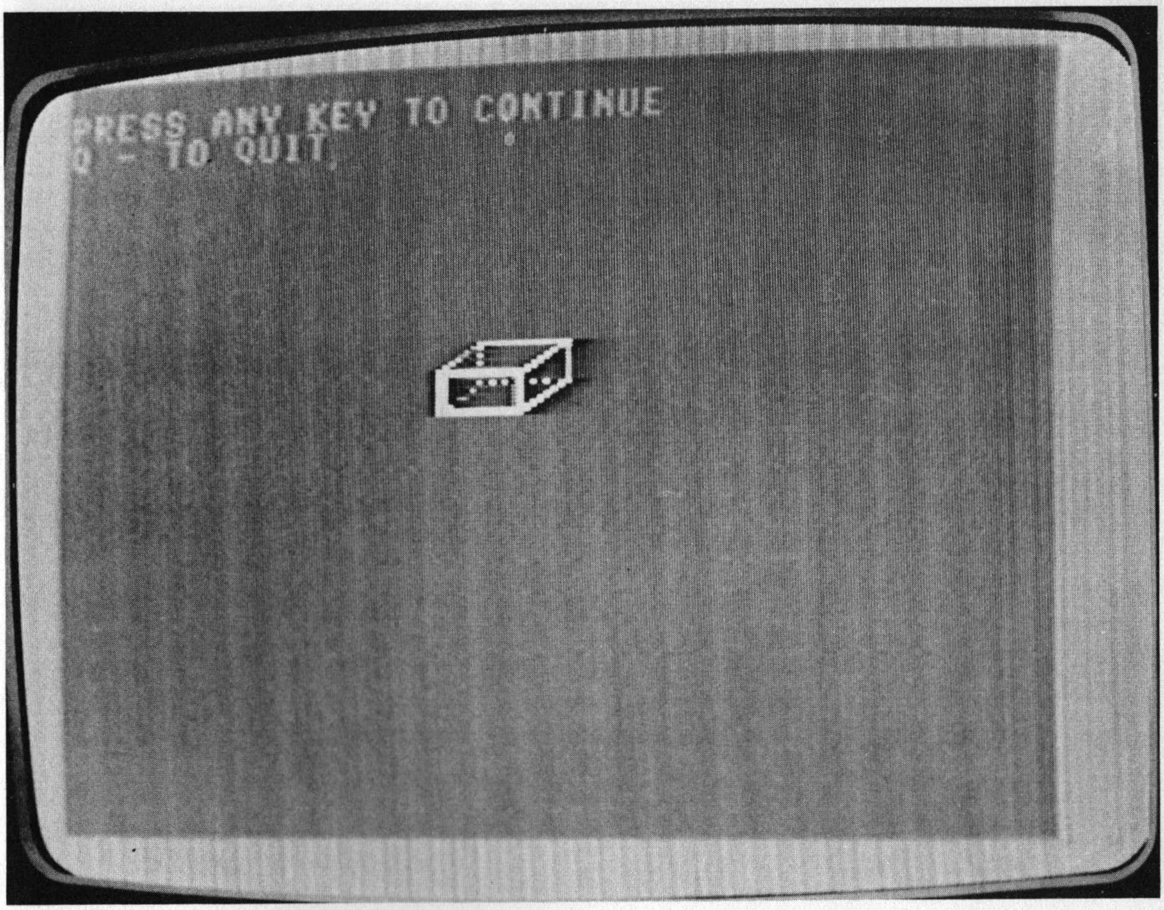
A sprite may be expanded without interfering with the current expansion status of other sprites by using the technique discussed for selectively enabling and disabling a sprite. For instance, if you wanted to expand sprite 4 along the X axis without changing the current status of sprite 2 or any other sprite, use the following statement:

```
100 POKE V+29,PEEK(V+29) OR 16
```

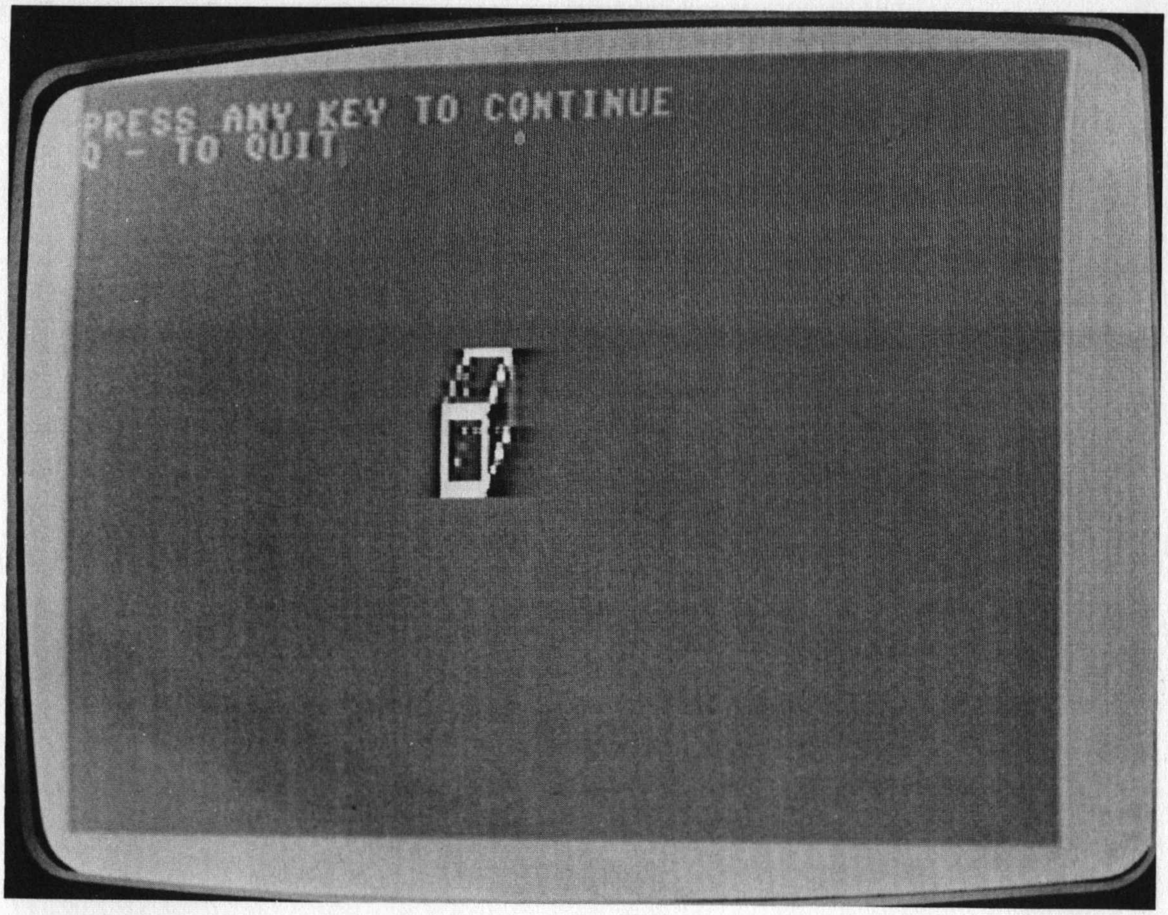
SELECTING SIZE AND COLOR IN THE SPRITE PROGRAM

In chapter 4, we wrote a program to select from one of five available sprites. Did you add your own sprites to this program? I hope so.

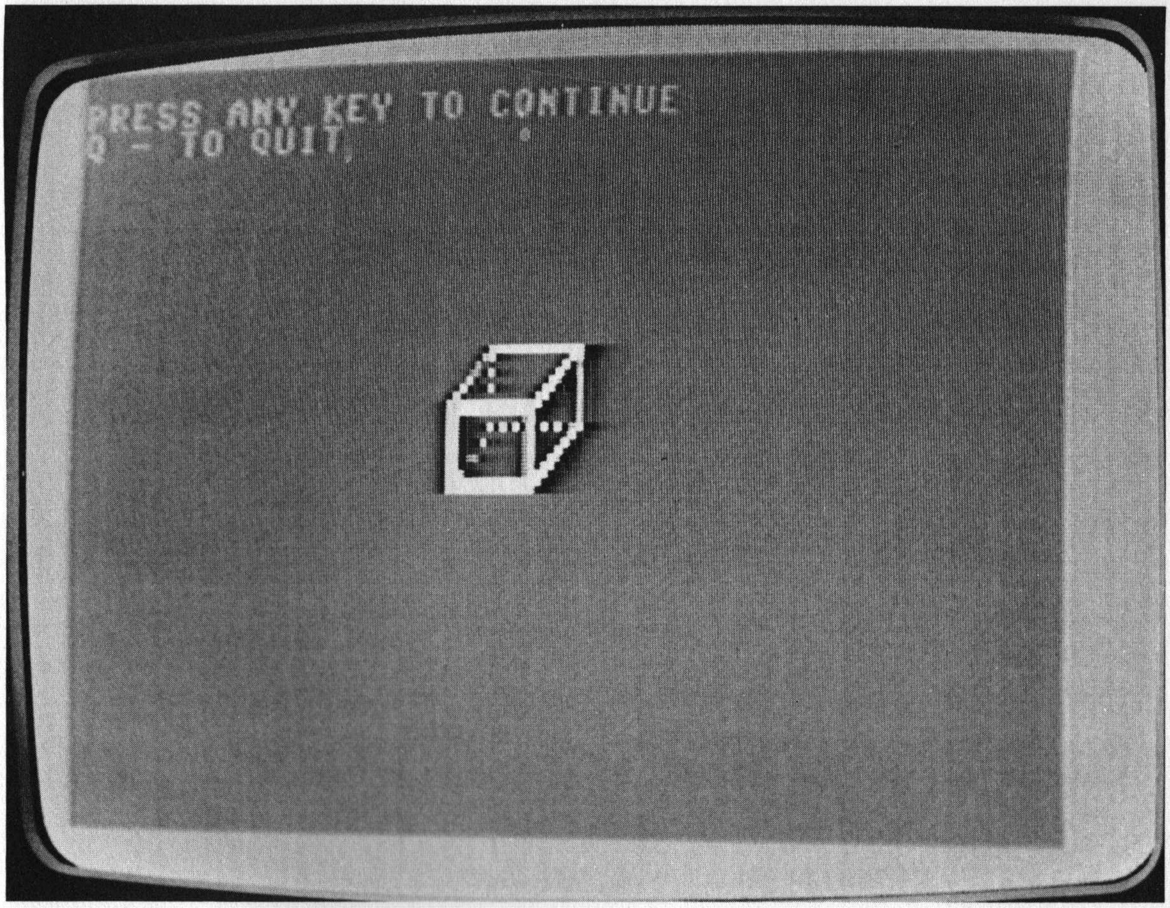
Now we are going to revise this program, so that the sprite size and color may also be selected by the user. After the sprite to be displayed has been selected, the program now asks whether expansion in the X or Y direction is required. It then proceeds to ask for the color you want for the sprite.



```
PRESS ANY KEY TO CONTINUE
Q - TO QUIT
```



```
PRESS ANY KEY TO CONTINUE
Q - TO QUIT
```



```
PRESS ANY KEY TO CONTINUE
Q - TO QUIT,
100 REM SELECT SPRITE
110 DIM SD(10,63)
120 V=53248
130 POKE 2042,195:REM SPRITE 2
140 FOR S=1 TO 5
150 FOR D=0 TO 62
160 READ SD(S,D)
170 NEXT:NEXT
180 GOSUB 500
190 GOTO 180
500 REM SELECT SPRITE
510 PRINT CHR$(147);TAB(12);"SELECT SPRITE"
520 PRINT
530 PRINT TAB(8);" 1 - SPACE SHUTTLE"
540 PRINT
550 PRINT TAB(8);" 2 - EAGLE        "
560 PRINT
570 PRINT TAB(8);" 3 - CAR          "
580 PRINT
590 PRINT TAB(8);" 4 - 3-D BOX      "
600 PRINT
610 PRINT TAB(8);" 5 - LETTER       "
620 PRINT
630 PRINT TAB(8);" 6 -              "
640 PRINT
650 PRINT TAB(8);" 7 -              "
660 PRINT
670 PRINT TAB(8);" 8 -              "
680 PRINT
690 PRINT TAB(8);" 9 -              "
700 PRINT
710 PRINT TAB(8);"10 -              "
715 PRINT
720 INPUT N
725 GOSUB 900
730 PRINT CHR$(147)
740 FOR I=0 TO 62:POKE 195*64+I,SD(N,I):NEXT
750 POKE V+4,150:POKE V+5,120
755 POKE V+21,4
760 PRINT "PRESS ANY KEY TO CONTINUE"
770 PRINT "Q - TO QUIT"
780 GET A$:IF A$="" THEN 780
790 IF A$="Q" THEN STOP
800 POKE V+21,0
810 RETURN
900 REM SELECT SIZE AND COLOR OF SPRITE
910 PRINT
920 EX=0:EY=0
930 INPUT "EXPAND IN THE Y DIRECTION (Y/N)";A$
940 IF LEFT$(A$,1)="Y" THEN EY=4
```

```
950 INPUT "EXPAND IN THE X DIRECTION (Y/N)";A$
960 IF LEFT$(A$,1)="Y" THEN EX=4
970 INPUT "ENTER COLOR CODE (0 TO 15)";C
980 POKE V+23,EY:POKE V+29,EX:POKE V+41,C
990 RETURN
1000 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,15
1010 DATA 0,0,30,3,0,60,4,255,248,15,255,240,31,255,252
1020 DATA 127,252,24,255,255,248,0,3,253,0,0,0
1030 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
1040 DATA 1,192,0,0,108,0,0,60,0,0,127,0,1,63,0,1,255,0
1050 DATA 0,255,0,1,255,128,0,127,128,0,255,128,0,127,128,
    0,63,240
1060 DATA 3,255,254,31,240,255,3,255,1,31,142,0
1070 DATA 16,10,0,0,10,0,0,0,0,0,0,0,0,0,0
1080 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
1090 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,3,252,0
1100 DATA 3,2,0,3,1,0,127,255,252,127,255,255
1110 DATA 247,255,223,99,255,142,0,0,0,0,0,0,0,0,0
1120 DATA 0,0,0,0,0,0,1,255,254,3,0,14,4,0,26,9,0,50
1130 DATA 16,0,98,33,0,194,64,1,130,255,255,2,255,254,2
1140 DATA 192,6,2,193,86,166,192,6,12,194,6,24
1150 DATA 192,6,48,200,6,96,192,6,192,224,7,128
1160 DATA 255,255,0,255,254,0
1170 DATA 15,255,252,63,255,248,64,8,8,128,24,0,192,24,0,
    0,24,0
1180 DATA 0,16,0,0,48,0,0,48,0,0,48,0,0,63,192,0,63,128
1190 DATA 0,96,0,0,96,0,0,192,0,0,192,0,0,128,0
1200 DATA 1,0,0,3,0,16,6,127,224,15,255,192
```

Comments on the Program

The main change to this program is the addition of subroutine 900.

Lines	
900–990	Subroutine to accept and activate the color and sprite size options.
920	EX and EY are variables to define the expansion on the X or Y axis.
930–940	If expansion in the Y direction is desired, set EY to 4. The number 4 refers to the value representing sprite 2.
950–960	Same thing for expansion on the X axis.
970	Accepts color code.
980	Sets the Y and X axes and the sprite color.

Challenge

Modify the program to move the chosen sprite across the screen from left to right. For a real challenge ask the user what direction the sprite should be moved, and then use the animation sequences of program code to move in the chosen direction.

```
[illegible]
```

Comments on the Program

[illegible]

Code

[illegible]

Exercise

[illegible]

6

Advanced Sprite Graphics

SPRITE PRIORITIES

The availability of sprite priorities gives the animation of sprites a high level of realism, which is impossible to attain with other types of screen graphics. By allocating appropriate priorities, one sprite is able to pass in front of or behind another. Individual sprites may be directed to pass in front of screen background information or even behind the background.

The priority of sprites is based on the sprite number; the lower the sprite number the higher the priority. As you can see from the following chart, sprite 0 has the highest priority, while sprite 7 has the lowest. Background has a lower priority than all of the sprites.

Sprite 0	Highest priority
Sprite 1	
Sprite 2	
Sprite 3	
Sprite 4	
Sprite 5	
Sprite 6	
Sprite 7	Lowest sprite priority
Background	Lowest priority

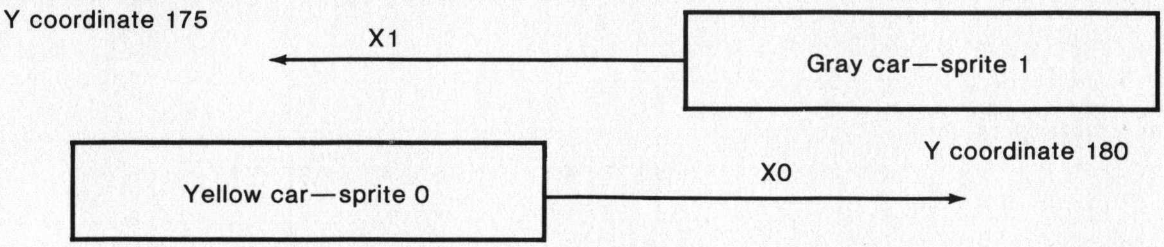
Sprites with a high priority will pass in front of a sprite with a lower priority. If sprite 0 and 1 were to cross paths on the screen, sprite 0 would pass in front of sprite 1. If sprites 1, 2, and 6 meet, sprite 2 will pass in front of sprite 6. But sprite 2 would pass behind sprite 1.

Obviously advance planning of your sprites and their numbers is essential in creating the effects you want. It will be necessary to decide which sprites should pass in front of or behind others when you design your program.

CARS PASSING ON THE HIGHWAY

In this program, the concept of sprite priorities will be applied to two cars that pass each other. A yellow car will be moved from left to right across the screen, while a gray car is moved in the opposite direction.

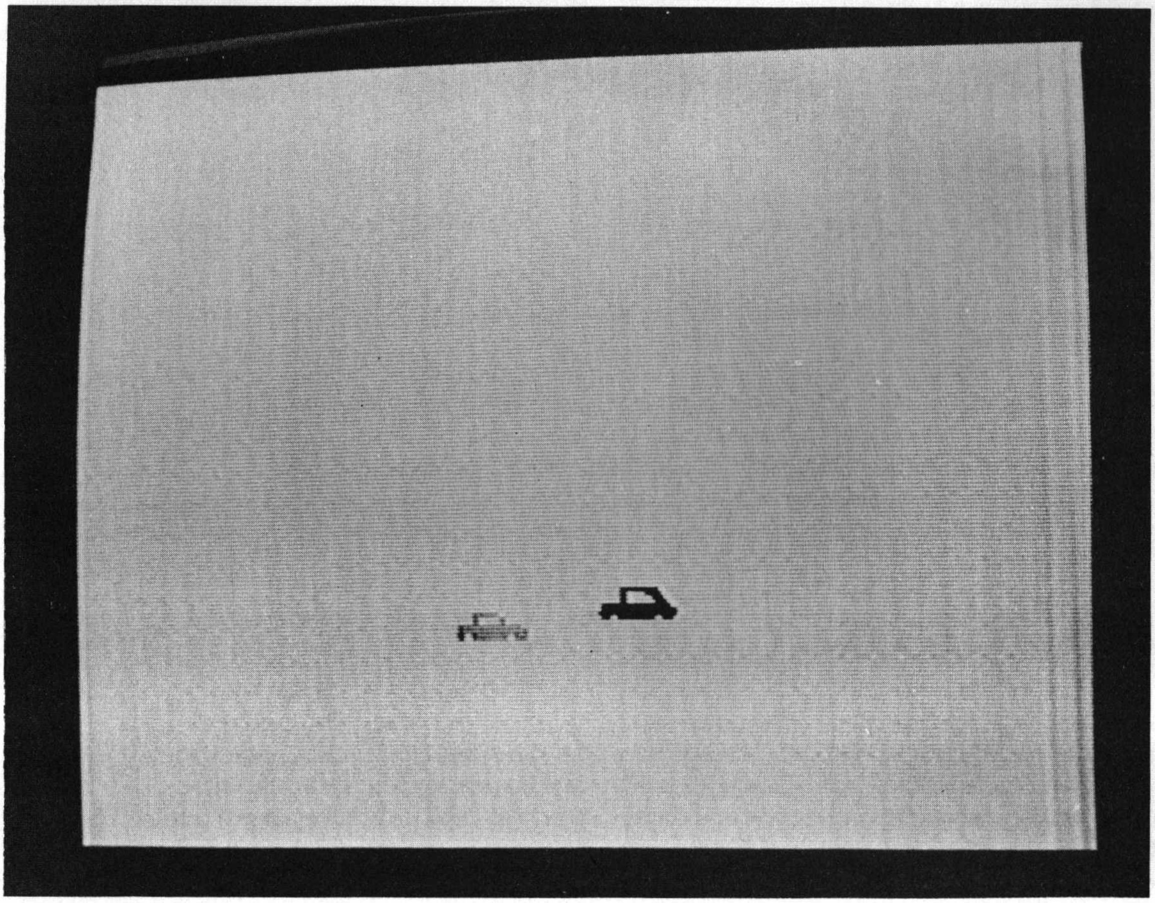
If you imagine these cars passing on a highway, the one coming from the left is closer to you, unless you are reading this in England or a country where the opposite driving pattern is used. Therefore the yellow car, which is closer, should have the higher priority—in this case sprite 0—and the gray car will be sprite 1.


Y coordinate 175
X1
Gray car—sprite 1
Y coordinate 180
X0
Yellow car—sprite 0

In addition to passing one sprite in front of the other, this program must animate two sprites in opposite directions. A formula for moving a sprite in the horizontal direction was given in the last chapter. This formula is used again in statements 250 and 260.

Notice the use of variables X0 and X1. These are used to define the position of the yellow car (sprite 0 uses X0) and the gray car (sprite 1 uses X1) along the X coordinate.

```
100 REM CARS ON HIGHWAY
105 CLR
110 PRINT CHR$(147)
115 POKE 53280,0:POKE 53281,1
120 V=53248
130 POKE 2040,192:POKE 2041,193
140 FOR S=0 TO 1
150 FOR I=0 TO 62
160 READ SD:POKE (192+S)*64+I,SD
170 NEXT:NEXT
175 POKE V+0,0:POKE V+2,88
```



```
180 POKE V+1,180:POKE V+3,175
190 POKE V+39,7:POKE V+40,11
200 POKE V+21,3:POKE V+16,2
210 FOR X1=343 TO 24 STEP -1
220 X0=343-X1
230 IF X1<256 THEN POKE V+16,PEEK(V+16) AND (255-2)
240 IF X0>255 THEN POKE V+16,PEEK(V+16) OR 1
250 POKE V+0,X0-(256*INT(X0/256))
260 POKE V+2,X1-(256*INT(X1/256))
270 NEXT
280 POKE V+21,0
1000 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
1005 DATA 0,0,0,0,0,0,0,0,0,0
1010 DATA 3,252,0,3,2,0,3,1,0,127,255,252,127,255,
   255,247,255,223
1020 DATA 99,255,142,0,0,0,0,0,0,0,0,0
1030 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
1035 DATA 0,0,0,0,0,0,1,255,192
1040 DATA 1,1,224,1,0,112,1,0,56,127,255,252,127,255,255,
   247,255,223
1020 DATA 99,255,142,0,0,0,0,0,0,0,0,0
```

Comments on the Program

Lines	
100–110	Clear memory and screen.
115	Sets white screen and black border.
120	Sets VIC-II chip address.
130	Sets pointers for sprite 0 and 1 data memory location.
140–170	Loops to read data and poke into memory.
175	Initializes the X position of the sprites off the visible screen.
180	Sets the Y coordinates.
190	Sets sprite 0 to yellow and sprite 1 to gray.
200	Enables sprites 0 and 1 (register 21) and sets the MSB for sprite 1.
210–270	Loop to animate sprites.
220	X0 is sprite 0's coordinate.
230	Resets MSB for sprite 1 at the border.
240	Resets MSB for sprite 0 at the border.
250–260	Sets the sprites' X coordinates.
280	Disables the sprites.

BACKGROUND TO SPRITE PRIORITY

The background has a lower priority than the 8 sprites, and so the sprites will normally pass in front of the background. This priority can be changed as we will see later.

Background can simply be a solid background color, which is what we have used so far in the sprite chapters, but you can also create screen graphics with either print statements or pokes and have the sprites pass in front of these images. Background can also contain text which may be desired in some programs.

CARS ON HIGHWAY REWRITTEN

The program with the two passing cars used an imaginary highway. Now let's revise that program to include a highway, a blacktop with a white centerline.

In planning this program, we need to consider the relationship between the highway and the cars. Obviously when we see cars driving along the highway, the cars are visible but the parts of the highway behind the cars are not. Therefore, the cars will need a higher priority than the road.

Since the highway will extend the full width of the screen, it makes sense that printing screen graphics will be a good way to create a road. Sprites are just not appropriate for that much space on the screen. Printing graphics also makes sense because these graphics will be part of the background, and the background has the required lower priority.

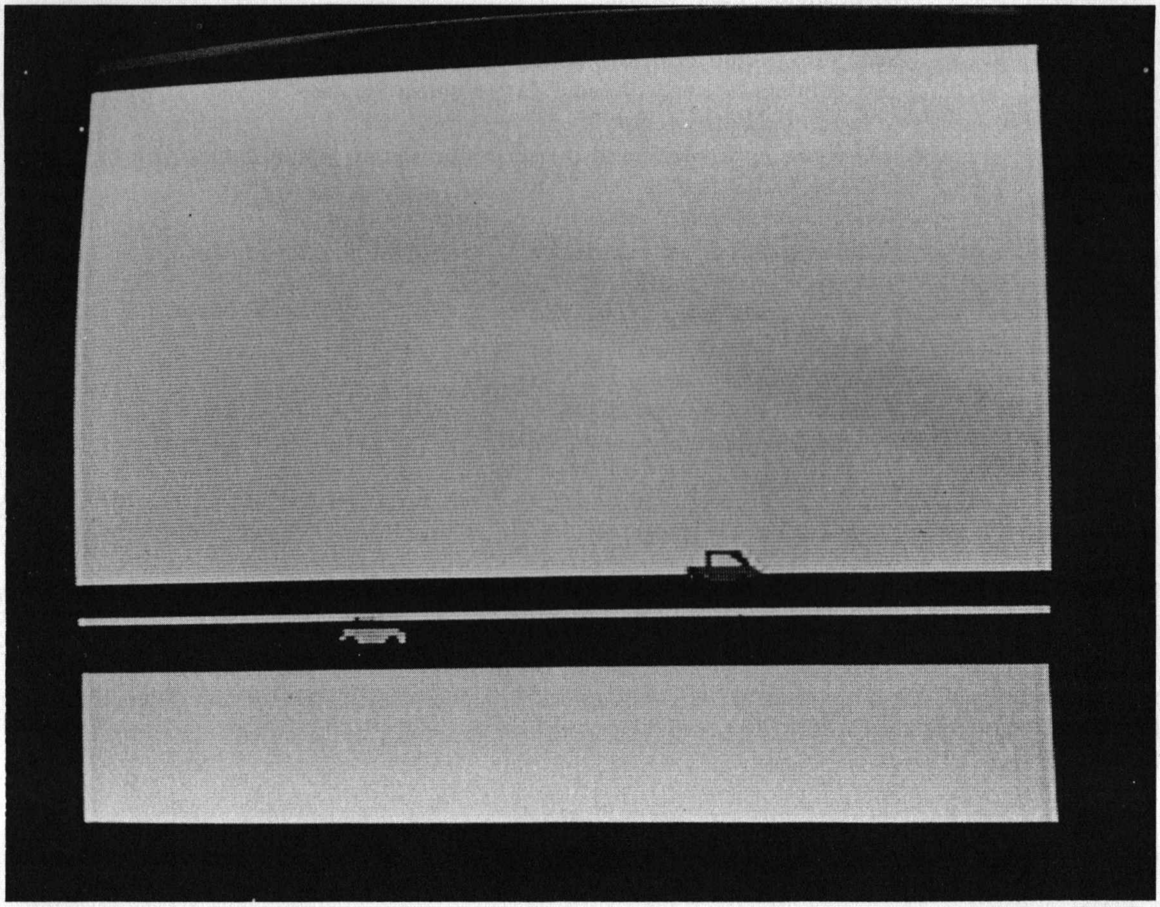
A few parts of the original program will need to be changed to create a background highway. One change, of course, is to print the highway itself. This is added as subroutine 300 which now clears the screen, positions the cursor, and prints the graphic characters for the highway.

To get the precise positioning of the cars on the highway, the correct combination of graphic characters, location of the lines, and the Y coordinates of the car, sprites must all be consistent. This can be a rather difficult task. But when you are creating a program like this, a little advanced planning can help. Then it might also help to experiment with a few different Y coordinate settings, until you get the effect you are looking for.

In the new program, Y coordinates 185 for sprite 0 and 170 for sprite 1 are used.

One other change. Now that there is a black highway, the gray car does not look too good with the black background. In fact, it is almost invisible. So sprite 1 has been changed to light blue (color code 14); I like a blue car better anyway.

This last change points out the importance of color choice as your graphics become more complex. With several different sprites, background graphics (which can use a variety of colors), and the screen's background color, conflicts in color choice can easily arise. So it is important to plan your colors carefully for the best effect.



```
100 REM HIGHWAY 2
105 CLR
110 GOSUB 300
115 POKE 53280,0:POKE 53281,1
120 V=53248
130 POKE 2040,192:POKE 2041,193
140 FOR S=0 TO 1
150 FOR I=0 TO 62
160 READ SD:POKE (192+S)*64+I,SD
170 NEXT:NEXT
175 POKE V+0,0:POKE V+2,88
180 POKE V+1,185:POKE V+3,170
190 POKE V+39,7:POKE V+40,14
200 POKE V+21,3:POKE V+16,2
210 FOR X1=343 TO 24 STEP -1
220 X0=343-X1
230 IF X1<256 THEN POKE V+16,PEEK(V+16) AND (255-2)
240 IF X0>255 THEN POKE V+16,PEEK(V+16) OR 1
250 POKE V+0,X0-(256*INT(X0/256))
260 POKE V+2,X1-(256*INT(X1/256))
270 NEXT
280 POKE V+21,0
300 REM DRAW LANDSCAPE
310 PRINT CHR$(147)
320 PRINT "{15 DOWNS}"
330 PRINT "{RVS}{BLK}{40 SPACES}";
340 PRINT "{RVS}{BLK}{40 E̲}";
350 PRINT "{RVS}{BLK}{40 SPACES}";
390 RETURN

1000 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
1005 DATA 0,0,0,0,0,0,0,0,0
1010 DATA 3,252,0,3,2,0,3,1,0,127,255,252,127,255,255,
    247,255,223
1020 DATA 99,255,142,0,0,0,0,0,0,0,0,0
1030 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
1035 DATA 0,0,0,0,0,0,1,255,192
1040 DATA 1,1,224,1,0,112,1,0,56,127,255,252,127,255,255,
    247,255,223
1020 DATA 99,255,142,0,0,0,0,0,0,0,0,0
```

Comments on the Program

Lines	
110	Go to the subroutine to display the highway graphic.
180	Sets new Y coordinates 185 and 170.
190	Sets colors yellow and light blue.
300–390	Subroutine to display the road and centerline.

Challenge

Modify the program to include a few buildings along the highway. It looks too much like the Great Salt Lake Desert. How about a house and a burger stand? Or maybe some mountains or trees? Use your imagination.

To make these changes, you will need fewer cursor downs in statement 320 to adjust for the number of lines you add to create your background.

BACKGROUND PRIORITY REGISTER

Having the background at a lower priority than the sprites is normally what you will want. Much of the time, activity on the screen occurs in front of the background which is there to make the screen more interesting. But the background does not usually take part in any screen action.

However, there are times when, for added realism, a sprite should pass behind the background. For instance, it would be interesting if an airplane could sometimes pass behind buildings instead of always in front of them, or a rocket could take off and disappear behind the clouds, which are part of a background image.

Well, by setting the background priority register V+27, you can give the background a higher priority than any selected sprite.

V=53248

Background Priority Register V+27

7	6	5	4	3	2	1	0	Sprite number
0	0	0	0	0	1	0	0	0—sprite higher 1—sprite lower than background
128	64	32	16	8	4	2	1	Decimal value

When a bit for a specific sprite is set to 1 in the background priority register, that sprite will have a lower priority than the background. In the diagram, sprite 2's bit is set to 1. As sprite 2 moves on the screen, it will be visible in front of the general screen background color. But when it encounters any screen graphics, such as a building, tree, cloud, etc., then sprite 2 will pass behind that object.

Since the other sprites have a background priority of 0, they would pass in front of any background images they encounter. So you can have some sprites pass in front of the background image and other sprites pass behind.

SUNSET

Here is another reason for having a sprite disappear behind the background. Did you ever watch the sun set over the ocean or a large lake? Then you know that it disappears behind the horizon. Of course, the same thing happens over land, but in this program the water will be the background.

The water will be produced with reverse spaces which are, what else, blue.

To write this program, we will need a sprite for the sun. To make things more interesting, the sun in the sky will be yellow. Then as it sets, it will turn orange, and, finally, before it sinks below the horizon, it will turn a deep red. At least it does on my television.

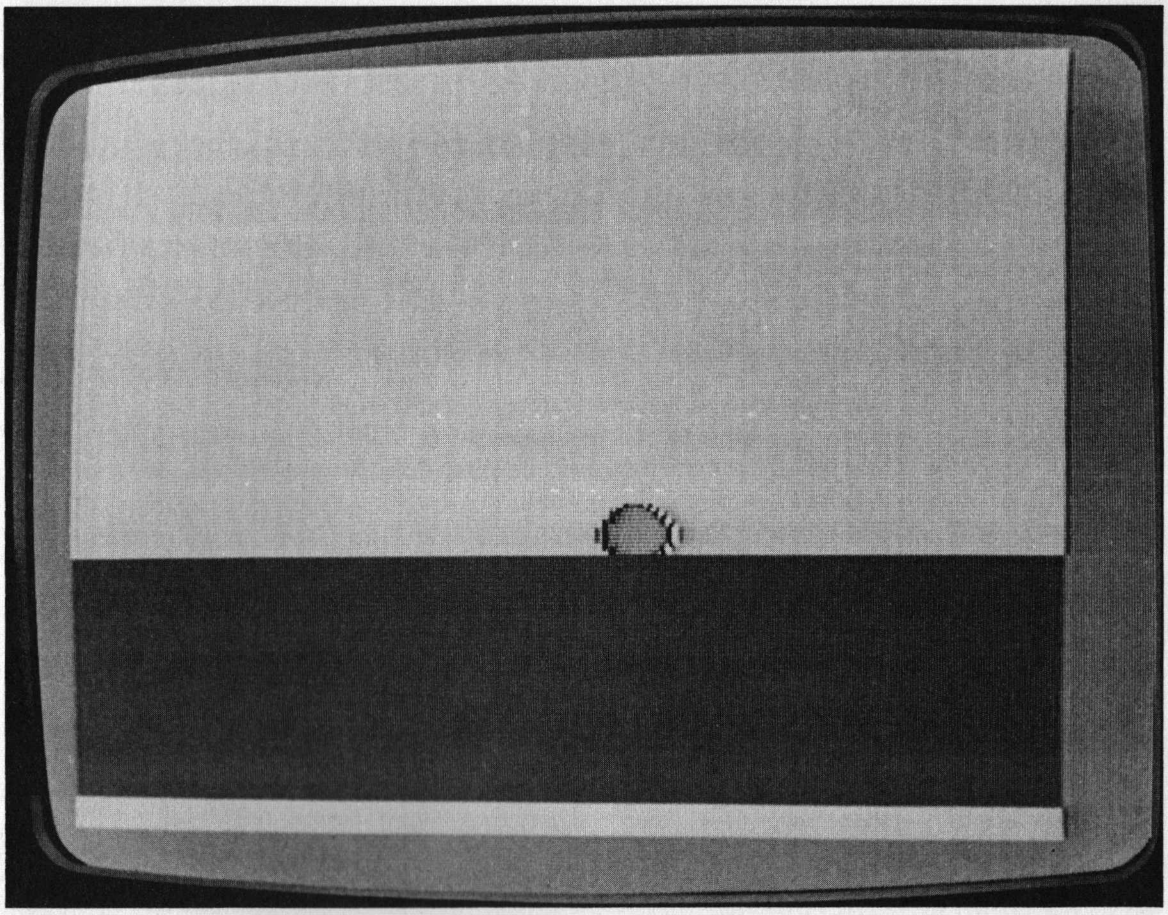
To move the sun, a loop to animate the sprite on the Y axis is needed. This is simple enough. But on the first run of the program, the sun seemed to just move below the horizon and the sunset was not very dramatic.

How do you add the dramatic to a sprite graphic program? Well, often a simple solution can be the most effective. How do you visualize the sun setting over the horizon? I like to think of it just hanging there in the sky like a big ball.

So, to create this effect, the speed of the sprite's movement is delayed. First there is a delay loop at line 150, which loops 50 times every time the sun is moved one pixel.

Now here's the rub. When the sun starts to set below the horizon (at Y coordinate 164), the delay loop's value is changed to 200 which slows the sun down considerably. The color is also changed to orange.

Next when the sun reaches Y coordinate 168, it is slowed down further, to a delay value of 300, and the color changes to red. This is a rather easy technique to include (slowing down the action), and I think you will agree that it is quite effective.



```
10 REM SUNSET 1
20 POKE 53280,14:POKE 53281,3
30 GOSUB 500
40 V=53248:N=50
50 POKE 2040,192
60 FOR I=0 TO 62
70 READ SD:POKE 192*64+I,SD
80 NEXT
90 POKE V+0,200
100 POKE V+39,7
110 POKE V+21,1
120 POKE V+27,1
130 FOR Y=50 TO 180
140 POKE V+1,Y
150 FOR D=1 TO N:NEXT
160 IF Y=164 THEN POKE V+39,8:N=200
170 IF Y=168 THEN POKE V+39,2:N=300
180 NEXT Y
190 STOP
500 REM WATER
510 PRINT CHR$(147)
520 PRINT "{15 DOWNS}";
530 FOR I=1 TO 8
540 PRINT "{BLU}{RVS}{40 SPACES}";
550 NEXT
```

```
560 PRINT CHR$(19)
570 RETURN
1000 DATA 0,0,0,0,255,0,7,255,192,15,255,240,31,255,248,
    63,255,252
1010 DATA 127,255,254,255,255,255,255,255,255,255,255,255,
    255,255,255
1020 DATA 127,255,254,63,255,252,31,255,248,15,255,240,
    7,255,192
1030 DATA 0,255,0,0,0,0,0,0,0,0,0,0,0,0,0
```

Comments on the Program

Lines	
20	Sets light blue border and cyan background (sky).
30	Go to subroutine to print the water.
40	Sets VIC-II chip address and initializes delay loop variable N.
50	Sets memory pointer for sprite 0.
60–80	Read sprite data into memory.
90	Sets X coordinate.
100	Sets sprite color to yellow.
110	Enables sprite 0.
120	Sets high background priority for sprite 0.
130–180	Loop to move sprite on the Y axis.
140	Defines sprite 0's position on the Y axis.
150	Delay loop.
160	At pixel position 164, set the color to orange and revise N to 200 to slow down the sunset.
170	At position 168, color becomes red and delay loop is set to 300.
500–570	Subroutine to display the water.
1000–1030	Data for the sun sprite.

Challenge

Can you find a graphic symbol to add some waves to the water? How about a few seagulls in the sky? Another interesting effect is to expand the sun in the Y direction just before it disappears over the horizon. Try it!

SPRITE COLLISION DETECTION

A technique many successful game programs use is to build the collision of two sprites into the strategy of the program. When a collision occurs between two or more sprites, bits representing the sprites are turned on in the sprite-to-sprite collision register, V+30, shown in the diagram below.

V=53248

Sprite-to-Sprite Collision Register V+30

7	6	5	4	3	2	1	0	Sprite number
0	0	0	0	0	1	1	0	0—no collision 1—sprite collision
128	64	32	16	8	4	2	1	Decimal value

When one sprite collides with another, the bit for each sprite involved is turned on. In the diagram, bits 1 and 2 are indicating sprites 1 and 2 have collided. This condition can be detected with a statement such as the following:

```
100 IF PEEK(V+30) AND 6=6 THEN 150
```

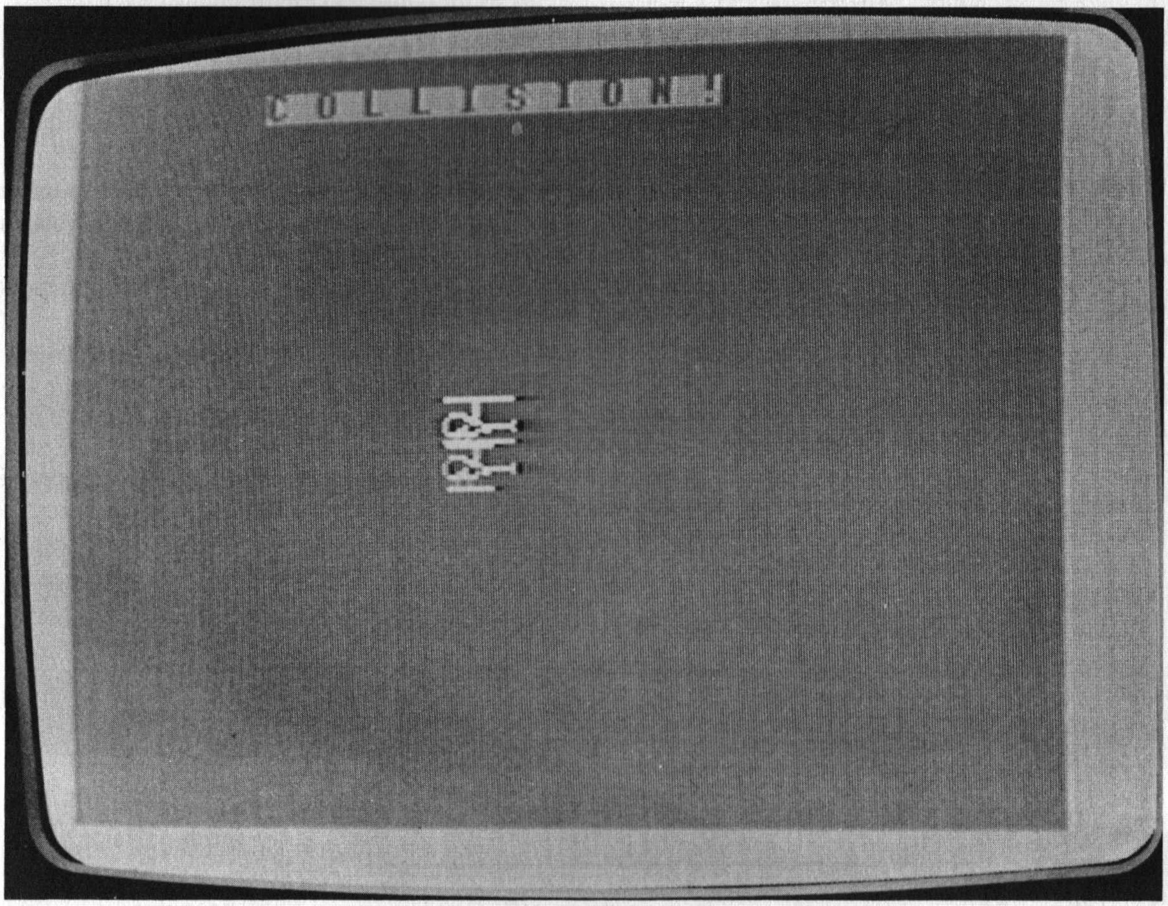
The value 6 in the above expression is the decimal value for sprites 1 and 2. When register V+30 is checked (with a PEEK), it is reset automatically to zero. The action in this example is to go to statement 150, but any action that is valid in the IF statement may be used.

CHOPPER COLLISION

In this program, a chopper is lifting off the tarmac while another is coming down. At a point near the center of the screen, the two collide and stop the action.

One consideration in this program, and any other program that uses collision detection, is that it is essential that the sprite-to-sprite collision register (V+30) is cleared before any action takes place. This can be done by an extra peek statement. In this program, it is statement 80.

```
10 REM CHOPPER COLLISION
20 PRINT CHR$(147)
30 V=53248
40 POKE 2040,192:POKE 2041,193
50 FOR D=0 TO 62:READ N:POKE 192*64+D,N:POKE 193*64+D,N:NEXT
```



```
60 POKE V+39,1:POKE V+40,3
70 FOR Y=51 TO 230
80 IF PEEK(V+30)AND 1=1 THEN 100
100 POKE V+1,Y:POKE V+0,150
110 POKE V+3,230-Y-49:POKE V+2,150
120 POKE V+21,3
130 IF PEEK(V+30) AND 3=3 THEN
    PRINT TAB(8)"{RVS}C O L L I S I O N !":STOP
140 NEXT Y
200 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,255,255,255
210 DATA 0,48,0,0,48,0,0,48,0,63,48,0,65,240,0
220 DATA 128,120,1,129,7,255,135,23,1,15.27,240,0
230 DATA 8,128,0,8,128,0,127,255,128,0,0,0,0,0,0,0,0,0
```

Comments on the Program

Lines	
20	Clears screen.
30	Sets VIC-II address.
40	Sets memory addresses for sprites 0 and 1.
50	Puts sprite data in memory.
60	Sprite 0 is white and sprite 1 is cyan.
70–140	Loop to animate sprites.
80	Dummy peek to reset register V+30.
100	Sets X and Y coordinates for sprite 0.
110	Sets X and Y coordinates for sprite 1.
120	Enables sprites 0 and 1.
130	Tests for collision of sprites 0 and 1. When they collide prints the message and stops the program.
200–230	Sprite data.

Challenge

Rewrite this program to reverse the direction of the helicopters when they hit. This is not as simple as it sounds and will require a different approach to the loop (lines 70 to 140) that is currently in the program.

Try a program with sprites moving along the horizontal axis. When they collide have them reverse direction. What impact does this have on the use of the MSB register?

SPRITE-TO-BACKGROUND COLLISION

Register V+31 can be used to detect collisions between a sprite and the background. This register is like the sprite-to-sprite, except that there is normally only one sprite that collides at a given time with the background. The diagram below shows the format for register V+31.

V=53248

Sprite-to-Background Collision Register V+31

7	6	5	4	3	2	1	0	Sprite number
0	0	0	0	0	1	0	0	0—no collision 1—sprite collision
128	64	32	16	8	4	2	1	Decimal value

To detect a collision with the background, a statement of the following type is used:

```
100 IF PEEK(V+31) AND 4=4 THEN 150
```

This statement would check for a collision between sprite 2 (value 4) and the background, as the one bit in the diagram indicates.

COLLISIONS OFF THE SCREEN

Sprites can collide at X and Y coordinates that are not visible screen locations. For example, a sprite located at coordinate 0,0 would be invisible but it can collide with another sprite that is in that region but is also invisible.

Depending on the type of program, this may or may not be a problem. If it is a problem, it may be necessary to check the coordinates before a specific action is taken when a collision occurs.

MULTICOLORED SPRITES

Sprites may be displayed in up to 4 colors, if you include the background color as part of the sprite. To set the multicolored mode, register V+28 is used to indicate which sprites are multicolored and which are a single color. The following diagram shows the register format for defining multicolored sprites.

Setting Multicolored Mode

A sprite which will be multicolored has its bit set to 1 in register V+28. You may include a combination of single and multicolored sprites in your programs. Just set the bits of each multicolored sprite to one, the same way you would enable the sprites.

To set sprite 2 to multicolor, as shown in the diagram, use the statement

```
POKE V+28,4
```

To set sprites 0, 2, and 5 to multicolored mode use the following poke statement:

```
POKE V+28,37
```

All other sprites will be single color by default, since their bits are set to 0.

V=53248

Sprite Multicolor Register V+28

7	6	5	4	3	2	1	0	Sprite number
0	0	0	0	0	1	0	0	0—single color sprite 1—multicolor sprite
128	64	32	16	8	4	2	1	Decimal value

Multicolor Registers

To define the colors used in multicolor mode, two new registers are used, in addition to the sprite color register and the screen background color. These new registers are the multicolor 1 and multicolor 2 registers located at V+37 and V+38, respectively.

Now all four colors are defined as follows:

Color	*Register*	*Bit Values in Sprite*
Background	53281	00
Multicolor 1	V+37	01
Sprite color	V+39	10
Multicolor 2	V+38	11

Each of these registers are set for color in the same way as we set the background or sprite colors in previous programs. For example, to set the background to black, the sprite color to yellow, multicolor 1 to light red, and multicolor 2 to light blue, the following statements could be used:

```
100 POKE 53281,0          Black background color
110 POKE V+37,10          Light red multicolor 1
120 POKE V+39,7           Yellow sprite color
130 POKE V+38,14          Light blue multicolor 2
```

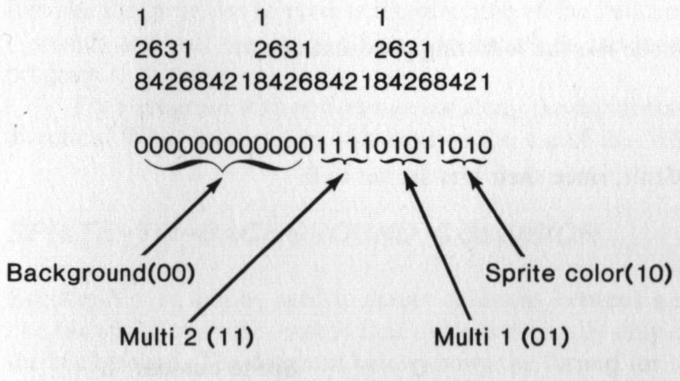
Whether these color combinations would work well together would depend on the sprite design and how the colors are placed next to each other. You may find that some experimentation with different color combinations for multicolored sprites will be necessary to get the right effect.

Multicolored Sprite Shapes

The bit values used in the preceding chart refer to the bits used to define a sprite's shape and color. Previously, sprite shapes were defined with ones to represent the image. With multicolored sprites, two bits are used for each sprite character. This has the effect of reducing the horizontal resolution by half.

Each bit in the sprite is represented by two bits. The value of the two bits can be one of 00, 01, 10, or 11. This value represents the color to be used for that part of the sprite.

For example, this sequence uses groups of 4 bits to represent each of the two background, multi 2, multi 1 and sprite character.

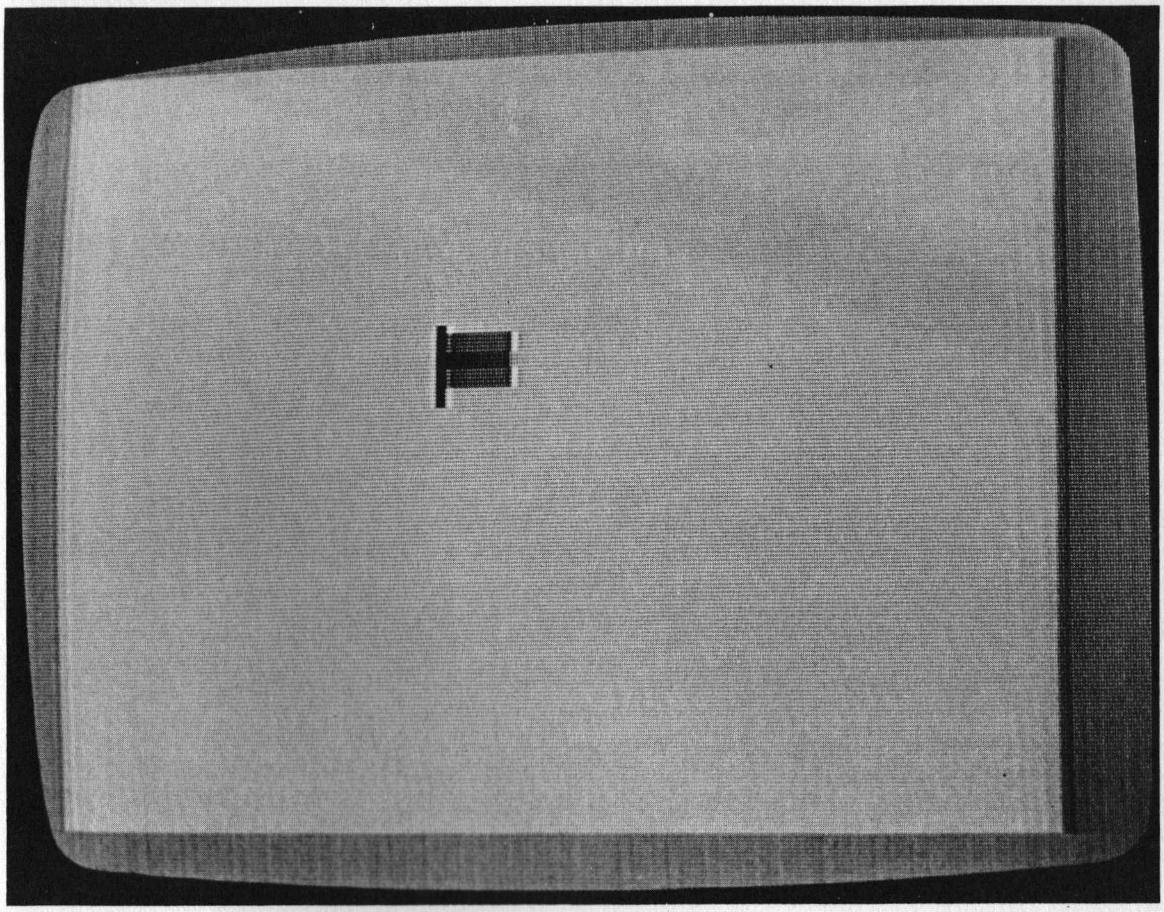


The data for this line is recorded as usual, using the decimal equivalent for the binary values.

```
DATA 0,15,90
```

CREATING A MULTICOLOR FLAG

The following program uses multicolor mode to create a flag with red and green horizontal stripes, a black pole, and a white background. I didn't check to see if this was the color and design used by some nation, but it does look rather pretty on the Commodore 64.



The color allocation was as follows:

Color	*Register*	*Bit Values*	*Color Code*
White Background	53281	00	1
Black Multicolor 1	V+37	01	0
Light Red Sprite color	V+39	10	10
Green Multicolor 2	V+38	11	5

Now the sprite design is created with these values on the following matrix.

```
100 REM FLAG
110 PRINT CHR$(147)
115 POKE 53280,7:POKE 53281,1
120 V=53248
125 POKE V+37,0:POKE V+38,5:POKE V+39,10:POKE V+28,1
130 POKE V+21,1:POKE 2040,192
140 FOR I=0 TO 62:READ D:POKE 192*64+I,D:NEXT
150 POKE V+0,150:POKE V+1,120
1000 DATA 64,0,0,64,0,0,106,170,170,106,170,170,106,170,170
1010 DATA 106,170,170,106,170,170,127,255,255,127,255,255,
127,255,255
1020 DATA 127,255,255,106,170,170,106,170,170,106,170,
170,106,170,170
1030 DATA 106,170,170,64,0,0,64,0,0,64,0,0,64,0,0,64,0,0
```

128	64	32	16	8	4	2	1	128	64	32	16	8	4	2	1	128	64	32	16	8	4	2	1	
0	1																							64, 0, 0
0	1																							64, 0, 0
0	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	106,170,170
0	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	106,170,170
0	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	106,170,170
0	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	106,170,170
0	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	106,170,170
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	127,255,255
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	127,255,255
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	127,255,255
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	127,255,255
0	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	106,170,170
0	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	106,170,170
0	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	106,170,170
0	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	106,170,170
0	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	106,170,170
0	1																							64, 0, 0
0	1																							64, 0, 0
0	1																							64, 0, 0
0	1																							64, 0, 0
0	1																							64, 0, 0

```
1000 DATA 64,0,0,64,0,0,106,170,170,106,170,170,106,170,170
1010 DATA 106,170,170,106,170,170,127,255,255,127,255,255,127,255,255
1020 DATA 127,255,255,106,170,170,106,170,170,106,170,170,106,170,170
1030 DATA 106,170,170,64,0,0,64,0,0,64,0,0,64,0,0,64,0,0
```

Comments on the Program

Lines	
115	Sets yellow border and white screen.
125	Sets multi 1 to black (V+37), multi 2 to green (V+38), sprite color to light red (V+39) and sets multicolor mode (V+28) for sprite 0.
130	Enables sprite 0 and sets memory pointer.
140	Loads sprite data.
150	Sets X and Y coordinates.
1000–1030	Multicolor flag sprite data.

Challenge

Create your own flag. How about some vertical stripes instead of horizontal? What other multicolored sprites are you interested in? Try some.

PART TWO

Sound and Music

7

The Basics of Music Generation

In the first part of this book, we examined a number of methods for producing exciting graphics on the Commodore 64. Now in this second part of the book, we are going to look at another fantastic capability hidden in the depths of the 64.

This new feature is the music and sound capability made possible by another chip in the computer, called the SID chip, which is short for Sound Interface Device.

To use SID effectively, we will need to learn about a number of techniques that relate to sound synthesis. In fact, SID is really a synthesizer, but don't let the small size fool you. SID is really a very capable performer, given the right instructions.

Some of the controls that are available for music synthesis are as follows:

1. Volume
2. Frequency
3. Waveform
4. Envelope generator—Attack/Decay/Sustain/Release (ADSR)
5. Note duration

There is more, but this will be more than enough for this chapter.

Obviously, sound is not quite as simple as 1, 2, 3, but it's not as difficult as you might suspect either. As with animating a sprite, there are several steps to learn when creating music, but each step is in itself quite simple.

Also keep in mind that learning the technical steps is one thing, but the ability to visualize a sprite and create it as you imagine it requires a different kind of ability. A very special part of this ability is simply persistence.

So it is with music. I'll teach you the steps for generating music, but, ultimately, you will need to work at developing just the right kinds of sounds that satisfy your ear.

GENERATING SOUND

Sound is produced in up to 3 different voices by poking memory locations to establish the different characteristics of the sound that you want to produce. To produce basic sounds you will need to set five kinds of registers: Volume, Attack/Decay, Sustain/Release, Frequency (actually two settings, a high frequency and a low one), and Waveform. Table 7.1 shows the addresses you will need for these registers.

Table 7.1 POKE addresses for SID sound settings

S=54272 **Type of Register**	**Voice 1**		**Voice 2**		**Voice 3**	
	Actual Address	**Program Usage**	**Actual Address**	**Program Usage**	**Actual Address**	**Program Usage**
Low Frequency	54272	V+0	54279	V+7	54286	V+14
High Frequency	54273	V+1	54280	V+8	54287	V+15
Waveform	54276	V+4	54283	V+11	54290	V+18
Attack/Decay	54277	V+5	54284	V+12	54291	V+19
Sustain/Release	54278	V+6	54285	V+13	54292	V+20
Volume	54296	V+24	54296	V+24	54296	V+24

Notice that the SID chip starts at address 54272 and the registers are addressed relative to this location in memory. Every sound program should use the statement

```
100 S=54272
```

to establish the starting address of the SID chip. Of course, the line number may be different depending upon the specific program.

VOLUME SETTING

The volume setting will determine how loud the sound will be. Values from 0 to 15 may be poked into register S+24 to set the sound level. Value 0 results in no sound and 15 is the maximum setting.

Since volume will usually be controlled by your monitor or sound system (yes, your Commodore 64 can be attached to a sound system for even more spectacular effects), a setting of 15 is most common.

Set the volume with a statement such as the following:

```
100 S=54272
110 POKE S+24,15
```

The first statement sets the SID address and statement 110 sets the volume to the maximum setting.

FREQUENCY SETTINGS

All sound is composed of sound waves that look like curves, when seen on an oscilloscope. A very pure tone would consist of a single curve such as a sine wave.

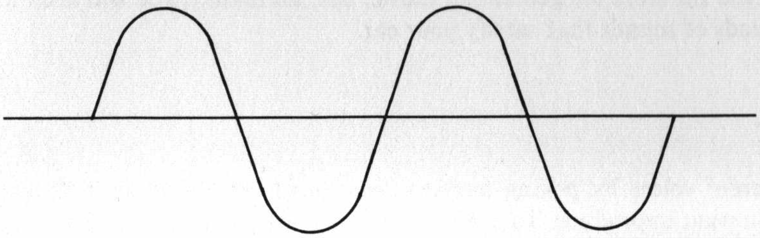


Table 7.2 POKE values for Octave Four Frequencies

S=54272

Voice Frequency POKE Values

Memory Addresses	**Voice 1**	**Voice 2**	**Voice 3**
High Frequency	S+1	S+8	S+15
Low Frequency	S+0	S+7	S+14

Notes in the Fourth Octave	C	C#	D	D#	E	F	F#	G	G#	A	A#	B	C	C#
High POKE	16	17	18	19	21	22	23	25	26	28	29	31	33	35
Low POKE	195	195	209	239	31	96	181	30	156	49	223	165	135	134
Decimal	4291	4547	4817	5103	5407	5728	6069	6430	6812	7217	7647	8101	8583	9094

Most sounds are much more complex than this pure tone and have harmonics that add richness to the sound. We will look at the concept of harmonics when waveform is discussed.

How frequently the tone varies from side to side on the line determines its frequency, and the frequency of a note determines its pitch. A high frequency produces a higher note than a low frequency. A standard in music is the A note which has a frequency of 440 cycles per second. The frequency of a C is 262 cycles per second. Frequency can also be expressed in Hertz, so an A is 440 Hertz.

The SID chip can generate sound in the range of approximately 16 to 3,952 cycles per second. Since orchestral music is generally in the range of 30 to 3,950 cycles per second, this gives our computer tremendous potential. It should be mentioned here that the harmonics produced by the instruments of the orchestra will be up to the limits of human hearing, about 18,000 cycles per second.

To set the frequency of the sound both the high and low frequency registers must be poked with a value to represent the note required. Table 7.2 shows the numbers to use for octave four. SID can produce eight octaves of sound, octaves 0 to 7. For a complete list of poke values see the appendix. For example, to set the frequency values to play a C in octave four, you would use the following two pokes.

```
120 POKE V+1,16          Set the high frequency for C
130 POKE V+0,195         Set the low frequency for C
```

To play an F♯ in the fourth octave, these statements could be used.

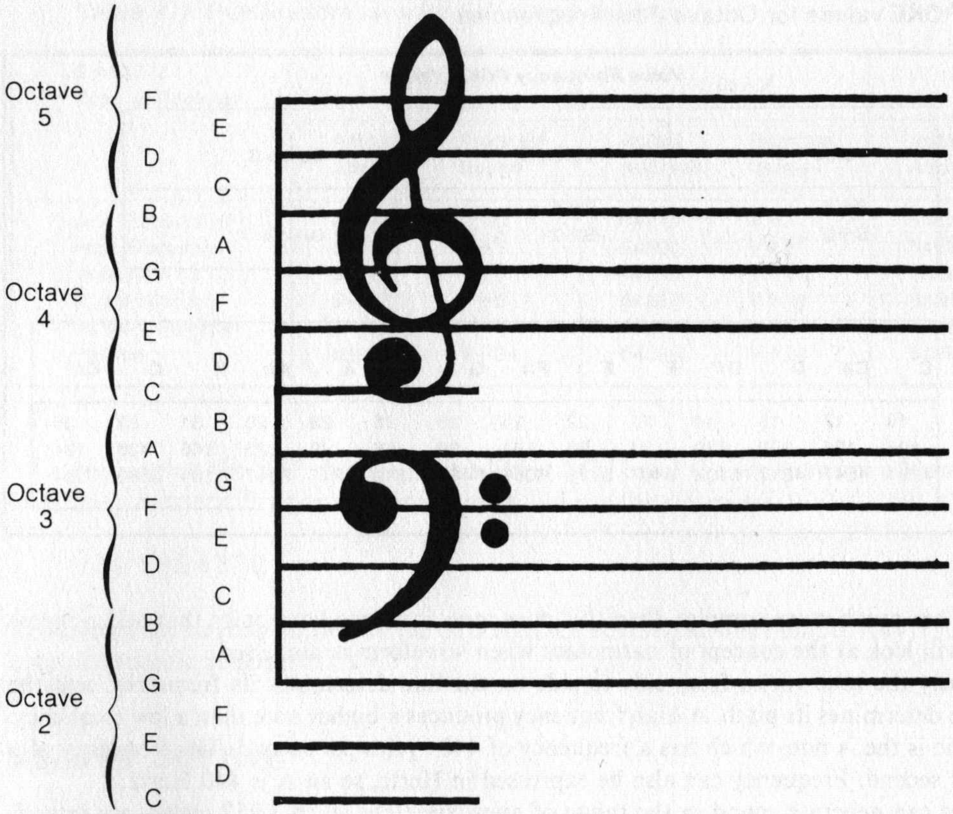
```
120 POKE V+1,23          Set the high frequency for F#
130 POKE V+0,181         Set the low frequency for F#
```

The note C at the right of the chart is the first note in the 5th octave. So if you used values 33 and 135, SID would play C, one octave higher than the fourth octave values.

MUSIC OCTAVES

Notes produced by the SID chip can be compared to notes on the treble and bass clefs of a musical score. These two clefs cover part of octave 2, all of octaves 3 and 4, and part of octave 5.

The following chart shows these relationships.



DATA STATEMENTS FOR NOTE VALUES

Poking values for the notes into frequency registers, as we have done, would not work too well for playing a song. We would need two pokes for each note in the song and, therefore, need to use an excessive number of pokes even for a relatively short song.

Instead, the note values are usually recorded in data statements and read by the program. These values can then easily be poked into the registers, as they are needed.

If we wanted to play a C followed by a D, then an E, the following data statement would be prepared:

```
1000 DATA 16,195,18,209,21,31
```

The values 16 and 195 are the high and low pair for C, 18 and 209 for D, and 21 and 31 for E. These values can be read and poked into the frequency registers with the following program code. Note that this is not a complete program and will not yet produce sound.

```
120 READ H,L
130 POKE S+1,H
140 POKE S+0,L
150 GOTO 120
1000 DATA 16,195,18,209,21,31
```

One difficulty with this code is that the program just crashes when it runs out of data. To eliminate this problem, we use special data at the end, such as -1, which can be recognized by the program.

```
120 READ H,L
130 IF H=-1 THEN STOP
140 POKE S+1,H
150 POKE S+0,L
160 GOTO 120
1000 DATA 16,195,18,209,21,31,-1,-1
```

Since this program reads two values each time, the data will need to end with two −1 values.

"SURPRISE SYMPHONY" (HAYDN)

For our first sound program, this piece of music is fairly easy to implement. The first step is to take the music and translate it into poke codes for the frequency registers. I find it helpful to write down the notes as letters, and then under them, write the high and low frequency codes for the data statements.

The subscript numbers indicate the octave. All notes, unless otherwise indicated, are in the 4th octave.

C	C	E	E	G	G	E	E
16	16	21	21	25	25	21	21
195	195	31	31	30	30	31	31
F	F	D	D	B_3	B_3	G_3	G_3
22	22	18	18	15	15	12	12
96	96	204	204	210	210	143	143
C	C	E	E	G	G	E	E
16	16	21	21	25	25	21	21
195	195	31	31	30	30	31	31
C_5	C_5	F#	F	G	G	G_3	
33	33	23	22	25	25	12	
135	135	181	96	30	30	143	

When the poke codes have been determined, these can be entered into data statements to be used in the program. The following program includes these codes in lines 1,000 to 1,030. If you run the program it will play the first four bars of Haydn's "Surprise Symphony." It should be clear from this example, that to record an entire song or piece of music, a lot of data will be needed.

For this reason most of the examples given will be shortened versions of the original. In many cases, the music will be rearranged so it will be suited to the needs of the computer. In the "Surprise Symphony," for example, all of the half notes were changed to two quarter notes. Later we will see how to play half notes, or full notes, or whatever duration you want.

```
10 REM SURPRISE SYMPHONY
20 FOR C=54272 TO 54296:POKE C,0:NEXT
30 S=54272
40 POKE S+24,15
50 POKE S+5,0
60 POKE S+6,248
70 READ N1,N2
80 IF N1=-1 THEN POKE S+24,0:STOP
90 POKE S+1,N1:POKE S+0,N2
```

```
100 POKE S+4,33
110 FOR I=1 TO 200:NEXT
120 POKE S+4,32
130 GOTO 70
1000 DATA 16,195,16,195,21,31,21,31,25,30,25,30,21,31,21,31
1010 DATA 22,96,22,96,18,209,18,209,15,210,15,210,12,
    143,12,143
1020 DATA 16,195,16,195,21,31,21,31,25,30,25,30,21,31,21,31
1030 DATA 33,135,33,135,23,181,22,96,25,30,25,30,12,
    143,-1,-1
```

Comments on the Program

Lines	
20	This loop clears all of the SID registers. It is a good idea to use this line at the beginning of every sound program.
30	Sets the address of the SID chip.
40	Sets the volume to the maximum setting.
50–60	These pokes set the ADSR registers. They will be discussed in detail later in the chapter.
70	Reads the high (N1) and low (N2) frequency values.
80	Checks for the end of data (−1). When the end is reached pokes the sound register to 0 to ensure the sound is stopped. Then stops the program.
90	Pokes the high and low frequency registers.
100–120	These statements set the waveform and provide a delay for note duration. These operations will be explained later in the chapter.
130	Go back to read more data.
1000–1030	Data for the music.

Challenge

Take a piece of music and code some data statements to replace the ones in the above program. At this point choose a simple piece, preferably in the key of C, with no sharps or flats. If you have a song with only quarter notes so much the better.

CONTROLLING NOTE DURATION

The "Surprise Symphony," as played in the above program, consisted of only quarter notes. However, the music did originally have some half notes in it. To compensate for these half notes, the program used two quarter notes which did not give quite the same effect.

This problem can be solved by including another value in the data which represents the type of note. Since there can be several note durations in the music, the following arbitrary values will be used.

Type of Note	*Data Value*
Eighth	1
Quarter	2
Dotted quarter	3
Half	4
Whole	8

Now each note will be represented by three numbers in the data statement.

```
1000 DATA 16,195,2,18,209,2,21,31,4
```

This data represents a C with a 2 to indicate a quarter note, a D which is also a quarter note, and an E with a duration of 4 for a half note.

The note duration value will be used in the delay loop to determine the amount of time each note is played. The following code reads these duration values into variable D, which, in turn, is used as a factor of 100 to control the delay loop in statement number 110.

```
70 READ N1,N2,D
80 IF N1=-1 THEN POKE S+24,0:STOP
90 POKE S+1,N1:POKE S+0,N2
100 POKE S+4,33
110 FOR I=1 TO 100*D:NEXT
120 POKE S+4,32
130 GOTO 70
1000 DATA 16,195,2,18,209,2,21,31,4,-1,-1,-1
```

Since three data values are being read in line 70, there must also be three −1 values to indicate the end of the data.

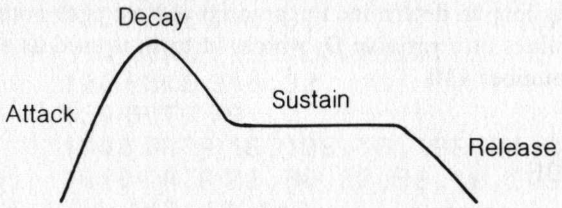
SECOND "SURPRISE SYMPHONY"

This technique gives the "Surprise Symphony" a smoother sound by incorporating half notes where they are needed in the music. If you prefer a faster or slower rendition of this music, the value 100 in statement 110 may be changed. A lower value will cause the music to play faster; a higher value plays more slowly.

```
10 REM SURPRISE 2
20 FOR C=54272 TO 54296:POKE C,0:NEXT
30 S=54272
40 POKE S+24,15
50 POKE S+5,0
60 POKE S+6,248
70 READ N1,N2,D
80 IF N1=-1 THEN POKE S+24,0:STOP
90 POKE S+1,N1:POKE S+0,N2
100 POKE S+4,33
110 FOR I=1 TO 100*D:NEXT
120 POKE S+4,32
130 GOTO 70
1000 DATA
16,195,2,16,195,2,21,31,2,21,31,2,25,30,2,25,30,2,
     21,31,4
1010 DATA 22,96,2,22,96,2,18,209,2,18,209,2,15,210,2,
     15,210,2,12,143,4
1020 DATA 16,195,2,16,195,2,21,31,2,21,31,2,25,30,2,
     25,30,2,21,31,4
1030 DATA 33,135,2,33,135,2,23,181,2,22,96,2,25,30,4,12,
     143,2,-1,-1,-1
```

ATTACK AND DECAY SETTINGS

The attack and decay register setting combined with the sustain and release register provide what is called the ADSR characteristic of the sound. Maybe the following chart will help explain what these characteristics are all about.



This chart indicates what happens to every note that is played. For example, when a note is struck on a piano, it first sounds quite loud; this is the attack which is the rate at which the sound reaches its highest or maximum level.

Then the volume of the note decays briefly. The decay is the rate at which the sound falls from the maximum level to a sustain level. It is then held at the sustain level until finally the key is released, and the note quickly fades away. This is the release characteristic.

Each instrument has a unique ADSR pattern. The piano has a very fast attack and decay, whereas a violin string that is bowed has a gradual attack, a very long sustain, and a gradual release pattern. The same string that is plucked will have an almost instantaneous attack which gives it a quite different sound.

Table 7.3 shows the poke values used for setting the attack and decay characteristics. Since there are three voices that can be played, these values can be poked into three different addresses depending on the voice(s) used. It is possible for each voice to be assigned a different characteristic.

Attack/decay values are formed by adding together the values required from both sides of the chart. For instance, a fast attack and a fast decay could be coded with 8, the sum of 0 + 8. This value would be poked into register 5 for voice 1 as follows:

```
200 POKE S+5,8
```

A slow attack might be formed with the values 64 + 32 and a fast decay with the values 4 + 2 + 1. The value to be poked is then the sum of all these (64 + 32 + 4 + 2 + 1), which is 103.

```
200 POKE S+5,103
```

It may take some experimentation with these values to get exactly the sound you are looking for in your music programs. But the complete sound characteristic must also include correct shaping of the Sustain/Release setting.

Table 7.3 Values for Attack/Decay Settings

S=54272	*Attack/Decay POKE Values*		
Memory Addresses	**Voice 1**	**Voice 2**	**Voice 3**
Program Uses	S+5	S+12	S+19
	Attack Value	**Decay Value**	
Slow Attack	128	8	*Fast Decay*
	64	4	
	32	2	
	16	1	
Fast Attack	0	0	*Slow Decay*

Table 7.4 Values for Sustain/Release Settings

S=54272	***Sustain/Release POKE Values***		
Memory Addresses	**Voice 1**	**Voice 2**	**Voice 3**
Program Uses	S+6	S+13	S+20
	Sustain Value	**Release Value**	
Long Sustain	128	8	*Slow Release*
	64	4	
	32	2	
	16	1	
Short Sustain	0	0	*Fast Release*

SUSTAIN AND RELEASE SETTINGS

Sustain and release refer to the characteristics of the sound after it has decayed to the sustain level and then fades out when the note is released. Sustain and release values are created and poked in the same way as the Attack/Decay settings. The total shaping of the sound depends upon the all four settings.

Table 7.4 shows the values and their meaning for the sustain and release settings of the ADSR registers.

A note played on a piano would have a long sustain and a fast release. This could be represented by values such as 128 + 0. The setting would be as follows:

```
210 POKE S+6,128
```

A shorter sustain could be set with the values 64 + 32 + 16 and a slower release with 4 + 1. The setting now would be 64 + 32 + 16 + 4 + 1 = 117.

```
210 POKE S+6,117
```

COMBINING ATTACK/DECAY AND SUSTAIN/RELEASE SETTINGS

The complete sound shape of an instrument is formed by all values of the ADSR registers. A tone might be created with an attack/decay value of 144 and a sustain/release of 247. These values were derived from the following pattern:

Attack	*Decay*		*Value*
128 + 16	0	=	144
128 + 64 + 32 + 16	4 + 2 + 1	=	247

Two pokes are needed to set these values in the SID memory.

```
200 POKE S+5,144: POKE S+6,247
```

Sometimes it is useful to use variables for these settings, so that they can be easily changed or even read from a data statement. The following code shows how to use variables for the ADSR values.

```
100 AD=144: SR=247
     .
     .
200 POKE S+5,AD: POKE S+6,SR
```

The value of 144 is assigned to AD for the attack and decay values, while 247 is assigned to SR for sustain and release. This action is taken at the beginning of the program. When the ADSR registers need to be set, two pokes are used, which get their values from the variables AD and SR.

"TAKE ME OUT TO THE BALL GAME"

Here's a familiar song. At least it is for baseball fans. The next program plays a few bars of the song using techniques we have discussed thus far. Note the use of a variable for attack/decay and a second for sustain/release. The values are assigned in statement 20 as follows:

```
20 AD=  9:SR=  68
```

These values give a pleasant oboelike sound to the music. Now try changing line 20 to use the following values. Do this by listing the program and overtyping line 20 with the new numbers. Don't forget to press return after the new values have been typed and then RUN the program.

These ADSR values produce a very mellow tone. Like a cello?

```
20 AD=  5:SR=  6
```

The next numbers create an organ-like tone.

```
20 AD=144:SR=247
```

These are more staccato, like a xylophone.

```
20 AD=  8:SR=  8
```

Here is the program.

```
10 REM BALLGAME
20 AD=  9:SR=  68
30 FOR C=54272 TO 54296:POKE C,0:NEXT
40 S=54272
50 POKE S+24,15
60 POKE S+5,AD:POKE S+6,SR
70 READ N1,N2,D
80 IF N1=-1 THEN POKE S+24,0:STOP
90 POKE S+1,N1:POKE S+0,N2
100 POKE S+4,17
110 FOR I=1 TO D*100:NEXT
120 POKE S+4,16
130 GOTO 70
1000 DATA 16,195,4,33,135,2,28,49,2,25,30,2,21,31,2,25,
     30,4,18,209,8
1010 DATA 16,195,4,33,135,2,28,49,2,25,30,2,21,31,2,25,30,
     4,-1,-1,-1
```

Comments on the Program

Lines	
20	Values for ADSR registers.
30	Clears SID registers.
40	Stores SID address.
50	Sets volume to maximum.
60	Sets attack/decay register (S+5) and sustain/release register (S+6).
70–80	Read high and low frequency values and delay value. When the last value (−1) is read, turn off the volume and stop the program.
90	Pokes note values in the high and low frequency registers.
100–120	Delay loop is used to control note duration. Set and reset waveform will be discussed next in the chapter.

Challenge

If you haven't already done so, try using the other suggested values for line 20. Now create your own values for ADSR settings. What sounds can your settings produce? These numbers will require some thought and planning, so a review of how they are developed might be helpful at this time.

WAVEFORM SETTINGS

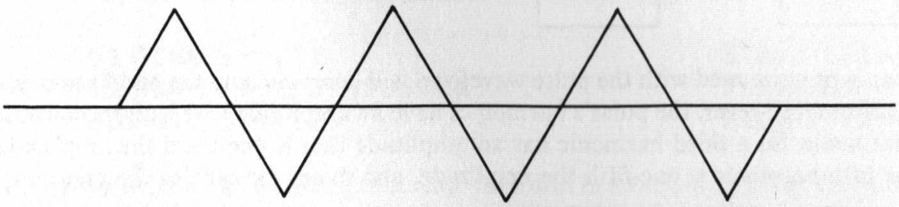
While ADSR determines one set of characteristics of the sound from a musical instrument, the waveform determines the coloring of the sound called the timbre. We have already seen how sound is composed of waves, such as a sine wave, and that the speed at which these waves vary determines the frequency of the sound.

Much of the uniqueness of the sound from each instrument is due to the timbre or kind of sound wave that the instrument creates. There are waves that resemble a triangle pattern, waves that are sawtooth, like the teeth of a saw, and waves that are square. In addition to the shape of the wave, the harmonics that are produced by an instrument also contribute to its unique sound.

Triangle Waveform

This waveform gives a soft, mellow, well-rounded tone. It might be compared to the sound produced by a flute, oboe, or French horn. The triangle waveform can also produce rich organlike tones.

A triangle waveform pattern is shown below.



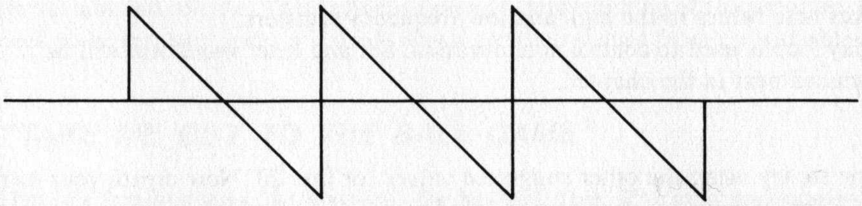
Triangle waveforms produce only odd harmonics. That is, the fundamental wave at the basic frequency of the tone is created. There is no second harmonic, but a second tone, called the third harmonic, is created at three times the fundamental tone. A fifth harmonic is produced, and a seventh, and so on.

Each subsequent harmonic has an amplitude that is equal to the reciprocal of the square of the harmonic's number. This means that the volume of the third harmonic is one-ninth as loud as the fundamental tone. The fifth harmonic is only one-twenty-fifth the volume of the fundamental, so, as you see, the harmonics quickly become inaudible.

Sawtooth Waveform

The sawtooth wave, as the name suggests, has the shape of the teeth on a saw. This gives it a bright, clear, brassy sound. Sawtooth waveforms can be used to simulate the sounds of brass instruments such as trumpets or bugles. It can also be given a fuller sound such as that produced by reed instruments: clarinets or saxophones.

Remember that the ADSR also contributes to the overall sound that the SID chip produces, and that the waveform only determines part of the effect. The sawtooth wave looks like this.



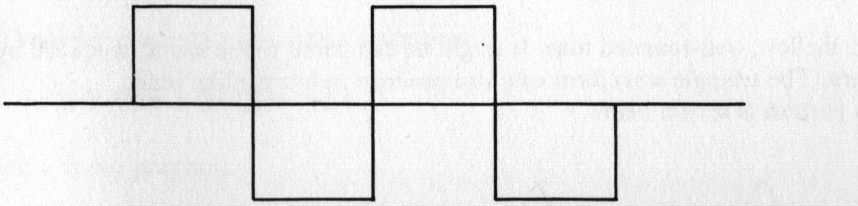
In a sawtooth wave all of the harmonics are present. As with the triangle waveform, these harmonics get progressively quieter but at a ratio that is equal to the reciprocal of the harmonic. The second harmonic will be half as loud as the fundamental tone, while the third harmonic will be one-third of the amplitude.

With all these harmonics present, the sawtooth is a sharp sounding tone which is clearly defined.

Pulse Waveform

The pulse waveform is unusual since you as the programmer have control over its shape. This is not true of the other waveforms. The pulse has a variable pulse width which is set by values in the program. As a result, the pulse form has great flexibility, but this also results in greater difficulty in its use.

The pulse is useful for simulating instruments, such as the piano or even a xylophone. Its waveform is shown by the following shape.



A square wave that is programmed with the pulse waveform will generate only the odd harmonics as does the triangle waveform. However, the pulse's harmonics have an amplitude that is the reciprocal of the number of the harmonic. So a third harmonic has an amplitude that is one-third the amplitude of the fundamental; the fifth harmonic is one-fifth the amplitude, and so on, for each of the odd-numbered harmonics.

Noise Waveform

The last waveform produces a type of white noise that has useful properties for special effects. Noise generates, at random, all of the frequencies and harmonics at equal amplitude. This lends the white noise generator to create the sounds of percussion instruments, or sounds for special effects not possible with the other waveforms.

Table 7.5 Values for the Waveform Settings

S=54272

Waveform POKE Values

Memory Addresses	**Voice 1**	**Voice 2**	**Voice 3**
Program Uses	S+4	S+11	S+18

	Start Value	**Stop Value**
Triangle	17	16
Sawtooth	33	32
Pulse	65	64
Noise	129	128

Table 7.6 Poke values for Pulse Rate

S=54272

Pulse Rate Values

Memory Addresses	**Voice 1**	**Voice 2**	**Voice 3**
High Pulse (0 to 15)	S+3	S+10	S+17
Low Pulse (0 to 255)	S+2	S+9	S+16

USING THE WAVEFORM SETTINGS

Table 7.5 shows the poke values for the waveform settings. There are two values for each waveform. The higher value, such as 17 for the triangle waveform, starts the vibration. Usually this poke is followed by the delay loop in the program. Then, after the appropriate delay, the waveform register is poked with the lower value (which is the start value minus 1) to stop the wave.

A common sequence of program code is the following:

```
100 POKE S+4,17                 Start triangle waveform
110 FOR I=1 TO D*100:NEXT       Time delay
120 POKE S+4,16                 Stop triangle waveform
```

Like the ADSR it can be useful to set the waveform in a variable, so that the value can be easily changed when needed. The following code uses a variable WF to supply the waveform value.

```
10 WF=17
   .
   .
100 POKE S+4,WF               Start triangle waveform
110 FOR I=1 TO D*100:NEXT     Time delay
120 POKE S+4,WF-1             Stop triangle waveform
```

When the pulse waveform is used, a pulse rate (table 7.6) must also be defined. This rate controls the width of the pulse and uses two registers—one for a high value and one for a low. These values are combined by the SID chip to define the total width of the waveform. Varying the high value will give the most audible effect.

The next three programs use each of the waveforms for a particular selection of music. Handel's "Largo" uses the triangle waveform, Rossini's "William Tell Overture" uses the sawtooth for a brassy effect, and the "Pizzicato Polka" uses the pulse waveform to give it a sound reminiscent of the xylophone.

"LARGO"

```
10 REM LARGO
20 AD=144:SR=247:WF=17
30 FOR C=54272 TO 54296:POKE C,0:NEXT
40 S=54272
50 POKE S+24,15
60 POKE S+5,AD:POKE S+6,SR
70 READ N1,N2,D
80 IF N1=-1 THEN POKE S+24,0:STOP
90 POKE S+1,N1:POKE S+0,N2
100 POKE S+4,WF
110 FOR I=1 TO D*100:NEXT
120 POKE S+4,WF-1
130 GOTO 70
1000 DATA 25,30,6,25,30,2,22,96,2,21,31,3,18,209,1,18,209,6
1010 DATA 21,31,2,22,96,2,25,30,1,28,49,1,31,165,1,28,49,6
1020 DATA 42,62,442,62,2,37,162,1,37,162,6
1030 DATA 33,135,4,33,135,3,31,165,1,31,165,6,-1,-1,-1
```

Comments on the Program

Lines	
20	Sets ADSR and triangle waveform values.
30–60	Initialize the SID registers.
70–130	Play the tune.
100–120	Start waveform, wait for the note delay, and then stop the waveform.

"WILLIAM TELL OVERTURE"

```
10 REM WILLIAM TELL
20 AD= 17:SR=130:WF=33
30 FOR C=54272 TO 54296:POKE C,0:NEXT
40 S=54272
50 POKE S+24,15
60 POKE S+5,AD:POKE S+6,SR
70 READ N1,N2,D
80 IF N1=-1 THEN POKE S+24,0:STOP
90 POKE S+1,N1:POKE S+0,N2
100 POKE S+4,WF
110 FOR I=1 TO D*100:NEXT
120 POKE S+4,WF-1
130 GOTO 70
1000 DATA 37,162,2,0,0,2,37,162,1,37,162,1,37,162
1005 DATA 4,0,0,2,37,162,1,37,162,1
1010 DATA 37,162,1,37,162,1,37,162,2,31,165,2,25,
     30,231,165,2
```

```
1020 DATA 31,165,2,37,162,2,31,165,2,37,162,2,50,60,2
1025 DATA 37,162,2,31,165,2,25,30,2
1030 DATA 31,165,2,37,162,2,31,165,2,37,162,2,47,107,2
1040 DATA 37,162,2,37,162,1,37,162,1,37,162,2,37,
    162,137,162,1
1050 DATA 37,162,1,37,162,2,37,162,2,37,162,2,-1,-1,-1
```

Comments on the Program

The "William Tell Overture" uses the same program as before except for changes in the attack/decay, sustain/release, and waveform. This program uses the sawtooth waveform. One other new feature is the use of 0,0,2 in the data. The zeros represent zero high and low frequency values which result in no sound being generated. In this music, these values are used for rests, and the number following the zeros gives the timing of the rest.

"PIZZICATO POLKA"

```
10 REM PIZZICATO
20 AD=  8:SR=  8:WF=65
30 FOR C=54272 TO 54296:POKE C,0:NEXT
40 S=54272
50 POKE S+24,15
60 POKE S+5,AD:POKE S+6,SR
70 READ N1,N2,D
80 IF N1=-1 THEN POKE S+24,0:STOP
90 POKE S+1,N1:POKE S+0,N2
100 POKE S+4,WF
110 FOR I=1 TO D*100:NEXT
120 POKE S+4,WF-1
130 GOTO 70
1000 DATA 21,31,1,25,30,1,18,209,1
1005 DATA 22,96,1,16,195,1,21,31,1,25,30,1
1010 DATA 28,49,1,33,135,1,31,165,2
1015 DATA 37,162,2,50,60,2,0,0,1,28,49,1
1020 DATA 33,135,1,25,30,1,31,165,1
1025 DATA 22,96,1,28,49,1,31,165,1,37,162,1
1030 DATA 44,193,1,42,62,2,33,135,2,33,135,2,-1,-1,-1
```

A program like this requires a lot of data for just a few bars of music. The reason, of course, is the extensive use of eighth notes. But, I believe you will find the result worth the extra typing effort.

Challenge

Experiment with these three pieces of music. Try different ADSR and waveform settings. What results can you produce? You can do just about anything with these settings, but notice that some effects just do not suit the music. Now enter some of your own favorite pieces. What settings work best with them?

8

More Music Techniques

Now that we have learned to use the basic features of synthesized sound on the Commodore 64, we will look at some more of its capabilities. In this chapter, we will learn to use the decimal frequency feature, which lets us change octaves quickly and easily. We will also learn to create multiple voice music, with two- or three-part harmony. Better still each part can be a different instrument.

USING DECIMAL FREQUENCY NUMBERS

Table 7.2 in the previous chapter showed the values to use for the high and low frequency registers. These values were used in all of the music programs created so far. One disadvantage to using the high and low values directly from the table is that two numbers must be typed for each note to be played.

Another problem with using these numbers is that to change octaves, it is necessary to retype all of the numbers with values for the new octave. There is no simple mathematical way to calculate the new high and low numbers for the octave you want.

But there is another method that can be used. This approach uses the decimal numbers provided in the chart. That chart is reproduced again as table 8.1.

The decimal values on the bottom line of table 8.1 represent the frequency of each note. For example, the number 4291 represents the note C in the fourth octave. To use this number, it must first be converted to a high and low poke value. You can do that with the following formulas.

High frequency = decimal frequency / 256
Low frequency = decimal frequency - (256 * High frequency)

In BASIC this conversion is done quite easily with two statements. Assuming the value 4291 was read into F, the following statements would produce the high and low frequency poke values.

```
100 H%=F/256
110 L%=F-(256*H%)
```

The variables H% and L% represent the high and low poke values. These are integer variables. Integers are necessary because these calculations will result in fractional values because of the division. But pokes can only use integer values.

If F contained 4291, then H% would get the value 16 and L% the value 195. These are the same high and low poke values the chart indicates for note C.

Table 8.1 POKE Values for Octave Four Frequencies

S=54272	***Voice Frequency POKE Values***		
Memory Addresses	**Voice 1**	**Voice 2**	**Voice 3**
High Frequency	S+1	S+8	S+15
Low Frequency	S+0	S+7	S+14

Notes in the Fourth Octave	C	C#	D	D#	E	F	F#	G	G#	A	A#	B	C	C#
High POKE	16	17	18	19	21	22	23	25	26	28	29	31	33	35
Low POKE	195	195	209	239	31	96	181	30	156	49	223	165	135	134
Decimal	4291	4547	4817	5103	5407	5728	6069	6430	6812	7217	7647	8101	8583	9094

RUNNING UP THE SCALE

Let's use this method now to do a short run up the scale in the fourth octave. The following program reads single values for each note from a data statement. Each value is converted to its corresponding poke values, and the appropriate registers are poked to produce the note.

The duration is fixed in the delay loop, but individual duration values could be read from the data as in previous programs. Try this program to confirm that this procedure does work.

```
10 REM SCALE 1
20 AD=18:SR=138:WF=33
30 FOR C=54272 TO 54296:POKE C,0:NEXT
40 S=54272
50 POKE S+24,15
60 POKE S+5,AD:POKE S+6,SR
70 READ F
80 IF F=-1 THEN POKE S+24,0: STOP
90 H%=F/256
100 L%=F-(256*H%)
110 POKE S+1,H%:POKE S+0,L%
120 POKE S+4,WF
130 FOR D=1 TO 200:NEXT
140 POKE S+4,WF-1
150 GOTO 70
200 DATA 4291,4817,5407,5728,6430,7217,8101,8583,-1
```

Comments on the Program

Lines	
20–60	Initialize SID registers.
70	Reads decimal frequency value.
80	Stops when end of music is reached.
90	Calculates high frequency poke value.
100	Calculates low frequency poke value.
110	Pokes high and low frequency registers.
120–140	Start and end sound with a delay loop between.

Challenge

Modify the scale program to play a short tune of your choosing. Unless all notes are of the same duration, you will need to add a duration value to the data, which is read by statement 70, and used in statement 130 to control the timing.

CHANGING THE OCTAVE

Using the decimal frequency value makes it easy to change octaves at will. This is handy since you will sometimes find that a particular tune sounds better in a particular octave. Changing octaves is also useful, if you want to repeat the tune several times but in a different octave each time.

To go up an octave simply multiply the original frequency value by two.

```
F=F*2
```

To go up two octaves multiply by three and so on.

```
F=F*3 (etc.)
```

To go down an octave divide by two.

```
F=F/2
```

Now insert the following statement in the program.

```
85 F=INT(F/3)
```

The INT function ensures that the result of dividing the value will be an integer. Dividing F by three will move the scale down two octaves, so now the program will play the scale in octave 2 instead of octave 4.

After you have tried the above statement, change it again to read

```
85 F=INT(F*2)
```

Actually the INT function is not strictly required this time since we are multiplying. Run the program and you will find the scale to be one octave above where it first began. That is, it will be octave 5 instead of octave 4.

REPEATING THE SCALE

Eight notes play through rather quickly, so it might be helpful to play the scale several times. Actually this technique can be modified to play parts of a melody where a bar or sequence of notes is played several times.

This time the program is modified, so that the reading and playing of each note is placed inside a FOR loop. The loop, which begins at statement 65, controls the number of times the scale is repeated.

At the end of each pass through the scale, the data must be restored. This is done with the RESTORE statement in line 160. The effect is to reset the computer to the beginning of the data, so the decimal numbers can be read again, and the scale played a second time, or a third time, or. . . .

```
10 REM SCALE 2
20 AD=18:SR=138:WF=33
30 FOR C=54272 TO 54296:POKE C,0:NEXT
40 S=54272
```

```
50 POKE S+24,15
60 POKE S+5,AD:POKE S+6,SR
65 FOR I=1 TO 4
70 READ F
80 IF F=-1 THEN 160
85 F=INT(F/4)
90 H%=F/256
100 L%=F-(256*H%)
110 POKE S+1,H%:POKE S+0,L%
120 POKE S+4,WF
130 FOR D=1 TO 200:NEXT
140 POKE S+4,WF-1
150 GOTO 70
160 RESTORE
170 NEXT I
180 POKE S+24,0
190 STOP
200 DATA 4291,4817,5407,5728,6430,7217,8101,8583,-1
```

Comments on the Program

Lines	
65	The FOR loop to line 170 repeats the scale four times. By changing the ending value on this line the scale may be played any number of times.
80	This line now branches to line 160 where the data pointer is restored.
85	Calculates the frequency in the octave desired.
160	Restores the data pointer.
170	End of loop beginning in line 65
180	Turns SID chip volume off.

Challenge

Try different octaves and change the number of repetitions of the scale. Add duration values so that the notes may each have different times: quarter, half, whole, etc.

STORING NOTES IN AN ARRAY

There will be times when you will want to repeat parts of a scale or even the whole thing. This can sometimes be handled by using the RESTORE statement and reading through the data a second time. Another method is to store the music data in an array and then make several passes through the array.

In the following program, the scale is first loaded into an array, and then the array is used to provide the data for the notes.

```
10 REM SCALE ARRAY
20 DIM F(8)
30 AD=18:SR=138:WF=33
40 FOR C=54272 TO 54296:POKE C,0:NEXT
50 FOR N=1 TO 8
60 READ F(N)
70 F(N)=INT(F(N)/4)
80 NEXT N
```

```
90 S=54272
100 POKE S+24,15
110 POKE S+5,AD:POKE S+6,SR
120 FOR I=1 TO 4
130 FOR K=1 TO 8
140 H%=F(K)/256
150 L%=F(K)-(256*H%)
160 POKE S+1,H%:POKE S+0,L%
170 POKE S+4,WF
180 FOR D=1 TO 200:NEXT
190 POKE S+4,WF-1
200 NEXT K
210 NEXT I
220 POKE S+24,0
230 STOP
1000 DATA 4291,4817,5407,5728,6430,7217,8101,8583
```

Comments on the Program

Lines	
20	Defines an array for the 8 decimal values.
50–80	Read the decimal frequency values into the array and adjust for the octave.
120–210	The I loop controls the number of times the scale will be played, in this case 4 times.
130–200	The K loop extracts each decimal value from the array, converts it to a high and low frequency value, and plays the note.

REVERSING THE SCALE

Once the data is in an array, it can easily be accessed in a different order. For example, if you want to play the scale from high to low just read the data from the end to the beginning of the array.

This can be done by changing line 130 to read

```
130 FOR K=8 TO 1 STEP -1
```

Challenge

Revise the scale program to play a scale from a low C to a high C and then reverse back down to the low C. This can be done in either two separate loops or use variables in the existing loop at 130, which is then revised as the program runs.

Another possibility with this program is to run the scale through several octaves by using only one octave of data. To do this recalculate the decimal frequency value after each pass of the array.

TWO- AND THREE-VOICE MUSIC

Producing two or even three part music is surprisingly easy on the Commodore 64. Recall that each of the charts in Chapter 7 had registers for attack/decay, sustain/release, and waveform. To play two or three parts, each of these registers must be poked with values to set each of the voices you want.

In addition, the high and low frequency registers must be poked for each voice. This action sets the tone for each note that is to be played simultaneously. Table 8.2 gives a summary of the addresses you will need.

Table 8.2 POKE Addresses for Multiple Voice Settings

S=54272 **Type of Register**	**Voice 1**		**Voice 2**		**Voice 3**	
	Actual Address	**Program Usage**	**Actual Address**	**Program Usage**	**Actual Address**	**Program Usage**
Low Frequency	54272	V+0	54279	V+7	54286	V+14
High Frequency	54273	V+1	54280	V+8	54287	V+15
Waveform	54276	V+4	54283	V+11	54290	V+18
Attack/Decay	54277	V+5	54284	V+12	54291	V+19
Sustain/Release	54278	V+6	54285	V+13	54292	V+20
Volume	54296	V+24	54296	V+24	54296	V+24

"AMAZING GRACE"

The old standard hymn is used to demonstrate the use of two-voice sound. Integer variables are used frequently to ensure that pokes are done with integer values, since the data will contain decimal frequency numbers.

For convenience, in this program each line of data will represent a melody note and its harmony, therefore, two-voice sound. The third value in each data statement is the delay value for the time duration of each note. Note that since the same pairs of notes are used frequently, typing can be reduced by overtyping each similar line with a new line number and pressing return.

```
10 REM AMAZING GRACE
20 A1= 18:S1=138:W1= 33
30 A2=144:S2=247:W2= 17
40 FOR C=54272 TO 54296:POKE C,0:NEXT
50 S=54272
60 POKE S+24,15
70 POKE S+5,A1:POKE S+6,S1
80 POKE S+12,A2:POKE S+13,S2
90 READ F1,F2,D
100 IF F1=-1 THEN POKE S+24,0:STOP
110 H1%=F1/256
120 L1%=F1-(256*H1%)
130 H2%=F2/256
140 L2%=F2-(256*H2%)
150 POKE S+1,H1%:POKE S+0,L1%
160 POKE S+8,H2%:POKE S+7,L2%
170 POKE S+4,W1
180 POKE S+11,W2
190 FOR I=1 TO D*150:NEXT
200 POKE S+4,W1-1
210 POKE S+11,W2-1
220 GOTO 90
1000 DATA 4817,4050,2
1010 DATA 6430,5407,4
1020 DATA 8101,6430,4
1030 DATA 6430,5407,4
1040 DATA 8101,6430,4
1050 DATA 7217,6069,2
1060 DATA 6430,5407,4
1070 DATA 5407,4291,1
1080 DATA 4817,4050,4
```

```
1090 DATA 4817,4050,2
1100 DATA 6430,5407,2
1110 DATA 8101,6430,1
1120 DATA 6430,5407,2
1130 DATA 8101,6430,4
1140 DATA 7217,6069,2
1150 DATA 9634,8101,6
1160 DATA 9634,8101,4
1170 DATA 8101,6430,2
1180 DATA 9634,8101,3
1190 DATA 8101,6430,1
1200 DATA 9634,8101,1
1210 DATA 8101,6430,1
1220 DATA 6430,5407,4
1230 DATA 4817,4050,2
1240 DATA 5407,4291,3
1250 DATA 6430,5407,1
1260 DATA 6430,5407,1
1270 DATA 5407,4291,1
1280 DATA 4817,4050,4
1290 DATA 4817,4050,2
1300 DATA 6430,5407,4
1310 DATA 8101,6430,1
1320 DATA 6430,5407,1
1330 DATA 8101,6430,4
1340 DATA 7217,6069,2
1350 DATA -1,-1,-1
```

Comments on the Program

Lines	
20	Sets ADSR and waveform values for voice 1.
30	Sets ADSR and waveform values for voice 2.
60	Volume setting for all voices.
70	Pokes voice 1's registers with ADSR values.
80	Pokes voice 2's registers with ADSR values.
90	Reads frequency values for voice 1 (F1) and voice 2 (F2).
110–120	Calculate voice 1's high and low frequency values.
130–140	Calculate voice 2's high and low frequency values.
150	Pokes the high and low frequency registers for voice 1.
160	Pokes the high and low frequency registers for voice 2.
170	Pokes voice 1's waveform.
180	Pokes voice 2's waveform.
1000–1360	Data.

Challenge

Take one of the programs from earlier in the book and add a second part to the music. As an alternate challenge write your own program in two-part harmony for one of your favorite songs. Sing along as it plays!

IMPROVING THE PROGRAM

Although the song "Amazing Grace" will be recognizable, there is obviously room for improvement. Although I'm not crazy about bagpipes, perhaps those of you with a little Scot in your blood would like the song to mimic the bagpipes.

Playing in a Higher Octave

One way to do this is to raise the octave, since pipes have a high pitched sound. This is easily accomplished by adding the line

```
105 F1=F1*2:F2=F2*2
```

This line places the music in octave 5. If you prefer an even higher sound, multiply by three instead of two in this line.

Getting a Smoother Tone

Now we are getting closer. But the bagpipe also has a smoother sound, whereas each note in this program is distinctly separate from the others. To let them blend, we should not turn the waveform off after each note is played.

Fix this by removing lines 200 and 210 from the program. Just type 200 and press return. Then type 210 and press return.

Adding the Drone

What is a bagpipe without the drone? The drone is the constant pitched sound that is heard along with the music as the pipes are played. Since we still have voice three available, it can be used to produce the drone.

This sound is quite easy to add, since it will be the same tone throughout the entire song. Therefore, we do not need to read a third note each time from the data statements. Instead we can just start the drone humming before the other pipes are played and let it continue droning until the song is finished.

To add the drone insert these statements:

```
85 POKE S+19,144:POKE S+20,128
86 POKE S+15,15 :POKE S+14,210
87 POKE S+18,33
```

Now the newly revised program will be as follows. The data is unchanged from the original program used above.

"AMAZING GRACE" WITH BAGPIPES

```
10 REM AMAZING GRACE WITH THE BAGPIPES
20 A1= 18:S1=138:W1= 33
30 A2=144:S2=247:W2= 17
40 FOR C=54272 TO 54296:POKE C,0:NEXT
50 S=54272
60 POKE S+24,15
70 POKE S+5,A1:POKE S+6,S1
80 POKE S+12,A2:POKE S+13,S2
85 POKE S+19,144:POKE S+20,128
86 POKE S+15,15 :POKE S+14,210
87 POKE S+18,33
90 READ F1,F2,D
100 IF F1=-1 THEN POKE S+24,0:STOP
105 F1=F1*2:F2=F2*2
110 H1%=F1/256
120 L1%=F1-(256*H1%)
130 H2%=F2/256
```

```
140 L2%=F2-(256*H2%)
150 POKE S+1,H1%:POKE S+0,L1%
160 POKE S+8,H2%:POKE S+7,L2%
170 POKE S+4,W1
180 POKE S+11,W2
190 FOR I=1 TO D*150:NEXT
220 GOTO 90
```

MULTIVOICE SOUND WITH INDEPENDENT DURATION

The last example showed how to do music with multiple voices, yet the duration of each note (except the drone) was the same. But much of the music you will want to play will have notes of different durations. This means one note will be finished before another. In fact, the different parts of the music may each have different timings.

The next program will solve this problem. It will permit the entry of three voices, and each voice will independently control its note duration.

Arrays are used in this program to store the music data. Table 8.3 shows the organization of these arrays. Each array is two-dimensional with three elements in the second dimension for the three-voice data. Array H% holds the high frequency value for each voice, L% holds the low frequency values, and D% holds the duration.

Each note is read as a decimal value and then converted with the formulas into the equivalent high and low frequency values, before being stored in the arrays. This process will take some time for a song the length of the "Skater's Waltz," so there will be a delay before the music starts to play.

Table 8.3 Arrays for Holding Multivoice Data

H% ***High Frequency Value***

		0	1	2	3	4	5	6	7 . . .	100
Voice 1	0									
Voice 2	1									
Voice 3	2									

L% ***Low Frequency Value***

		0	1	2	3	4	5	6	7 . . .	100
Voice 1	0									
Voice 2	1									
Voice 3	2									

D% ***Duration of Note***

		0	1	2	3	4	5	6	7 . . .	100
Voice 1	0									
Voice 2	1									
Voice 3	2									

The data consists of three groups, one for each voice. Each group of data ends with -1 values. The last group, for voice three, ends with -1 values to end the voice and -2 values to end the music.

Arrays are also used in this program to hold the attack/decay values (AD(2)), sustain/release (SR(2)), and waveform (WF(2)). Subscripts 0, 1, and 2 will refer to the positions for voices 1, 2, and 3, in that order.

Subroutine 400 plays the music. Variables N1, N2, and N3 are used as pointers to identify the note to be played for each voice. These variables are updated independently so that each voice can proceed at its own rate.

The time duration for each note is controlled by array D%. As each value is taken from this array, it is placed in variable D1, D2, or D3, depending on which voice it represents. These variables are each reduced by 1 when the delay loop is completed. One pass through the delay loop represents an eighth note. As either D1, D2, or D3 reaches zero, the waveform for that voice is turned off.

Now, the next note for that voice is poked into the high and low frequency registers. While this is happening, the other two voices will continue to play without interruption. If two or three voices end at the same time, the process happens quickly enough that the delay between resetting each voice will not be noticeable.

Here is the program.

"SKATER'S WALTZ"

```
100 REM SKATERS WALTZ
110 DIM H%(100,2),L%(100,2),D%(100,2)
120 AD(0)=18:SR(0)=138:WF(0)=33
130 AD(1)= 9:SR(1)= 68:WF(1)=17
140 AD(2)=17:SR(2)=130:WF(2)=33
150 FOR C=54272 TO 54296:POKE C,0:NEXT
160 S=54272
170 I=0:V=0
180 GOSUB 300
190 POKE S+24,15
200 POKE S+5,AD(0):POKE S+6,SR(0)
210 POKE S+12,AD(1):POKE S+13,SR(1)
220 POKE S+19,AD(2):POKE S+20,SR(2)
230 I=0
240 GOSUB 400
250 POKE S+24,0
260 STOP
300 REM READ MUSIC DATA
305 PRINT CHR$(147);TAB(15);"W A I T"
310 READ F,T
320 IF F=-2 THEN RETURN
330 IF F=-1 THEN N(V)=I-1:I=0:V=V+1:GOTO 310
340 H%(I,V)=F/256
350 L%(I,V)=F-(256*H%(I,V))
360 D%(I,V)=T
370 I=I+1
380 GOTO 310
400 REM PLAY MUSIC
410 GOSUB 500
420 GOSUB 600
430 GOSUB 700
440 FOR D=1 TO 80:NEXT
450 D1=D1-1:D2=D2-1:D3=D3-1
```

```
460 IF D1=0 THEN POKE S+4,WF(0)-1:GOSUB 500
470 IF D2=0 THEN POKE S+11,WF(1)-1:GOSUB 600
480 IF D3=0 THEN POKE S+18,WF(2)-1:GOSUB 700
490 IF N1=N(0) AND N2=N(1) AND N3=N(2) THEN RETURN
495 GOTO 440
500 REM VOICE 1
510 IF N1=N(0) THEN RETURN
520 POKE S+1,H%(N1,0):POKE S+0,L%(N1,0)
530 D1=D%(N1,0)
540 POKE S+4,WF(0)
550 N1=N1+1
560 RETURN
600 REM VOICE 2
610 IF N2=N(1) THEN RETURN
620 POKE S+8,H%(N2,1):POKE S+7,L%(N2,1)
630 D2=D%(N2,1)
640 POKE S+11,WF(1)
650 N2=N2+1
660 RETURN
700 REM VOICE 3
710 IF N3=N(2) THEN RETURN
720 POKE S+15,H%(N3,2):POKE S+14,L%(N3,2)
730 D3=D%(N3,2)
740 POKE S+18,WF(2)
750 N3=N3+1
760 RETURN
1000 DATA 5407,6,6430,4,7217,2
1010 DATA 7217,6,7217,6,5728,6,7217,6,8101,2
1020 DATA 8101,2,8101,2,9634,6,8583,4,5407,2
1030 DATA 6430,6,5728,4,5407,2,5407,6
1040 DATA 4817,6,4291,6,4291,4,0,1,-1,-1
1050 DATA 2145,2,2703,2,2703,2
1060 DATA 2145,2,2703,2,2703,2
1070 DATA 2025,2,2864,2,2864,2
1080 DATA 1607,2,2864,2,2864,2
1090 DATA 2408,2,2864,2,2864,2
1100 DATA 1607,2,2864,2,2864,2
1110 DATA 2145,2,2703,2,2703,2
1120 DATA 2145,2,2703,2,2703,2
1130 DATA 1804,2,3608,2,3608,2
1140 DATA 2703,2,3608,2,3608,2
1150 DATA 2408,2,3608,2,3608,2
1160 DATA 2408,2,3608,2,3608,2
1170 DATA 1607,2,2864,2,2864,2
1180 DATA 1607,2,2864,2,2864,2
1190 DATA 2145,2,2703,2,2703,2
1200 DATA 2703,4,0,1,-1,-1
1210 DATA 0,2,3215,2,3215,2
1220 DATA 0,2,3215,2,3215,2
1230 DATA 0,2,3215,2,3215,2
1240 DATA 0,2,3215,2,3215,2
1250 DATA 0,2,3215,2,3215,2
1260 DATA 0,2,3215,2,3215,2
1270 DATA 0,2,3215,2,3215,2
1280 DATA 0,2,3215,2,3215,2
```

```
1290 DATA 0,2,4291,2,4291,2
1300 DATA 0,2,4291,2,4291,2
1310 DATA 0,2,4291,2,4291,2
1320 DATA 0,2,4291,2,4291,2
1330 DATA 0,2,3215,2,3215,2
1340 DATA 0,2,3215,2,3215,2
1350 DATA 0,2,3215,2,3215,2
1360 DATA 3215,4,0,1,-1,-1,-2,-2
```

Comments on the Program

Lines	
110	Arrays for high and low frequency values and duration. A maximum of 101 notes can be stored.
120–140	Sets attack/decay, sustain/release, and waveform values.
160–180	Initialize variables for SID address, I for the position in the arrays of each music note and V for each voice. Subroutine 300 loads the music data into the arrays.
190–220	Sets SID registers for volume, and ADSR values.
240	Subroutine 400 plays the music.
250–260	Turn off the volume and stop the program.
300–380	This subroutine reads all of the music data into the arrays.
310	Reads the decimal frequency value and time duration.
320	A -2 means all of the data has been read, so return from the subroutine.
330	A -1 means the end of one part (notes for a single voice) has been reached. N(V) is now adjusted to hold the numbers of notes stored for that voice. I is reset to zero for the next voice, and the voice counter V is increased to the number for the next voice.
340–360	High and low frequency values and duration are calculated and stored in their respective arrays.
370	I is the note counter.
400–495	This subroutine plays the music. It also uses subroutines 500, 600, and 700 to poke the note values for each voice as it is needed.
410–430	Start the first note of each voice.
440	Delays loop for an eighth note.
450	Reduces each delay counter by 1. For example, if D1 had been a 2 initially, then the note for voice 1 was a quarter note. Since the delay loop plays for an eighth-note duration the quarter note will require two loops which are controlled by D1's value.
460–480	When a note's duration is reduced to zero the waveform is set off. Then the subroutine for that voice sets the values for the next note.
490	If all values are at the end of the array the music is finished and the program returns from the play music subroutine.
500–560	This subroutine gets one note for voice 1 and plays it.
510	Checks to see if the last note for voice 1 has been played.
520	Pokes the high and low frequency values for voice 1.
530	D1 takes on the delay value for the note.
540	Sets the waveform for voice 1.
550	Increments the note counter.
600–660	and
700–760	These subroutines follow the same pattern as subroutine 500 but are for voice 2 and voice 3.
1000–1040	Data for voice 1.
1050–1200	Data for voice 2.
1210–1360	Data for voice 3.

Challenge

The potential for challenges at this point is almost endless. For starters try to code some data for music of your choice.

Try using different settings for the ADSR and waveform. What effects can you produce with the three voices?

There are some inefficiencies in the program. For instance, subroutines 500, 600, and 700 are similar. Can you replace them with a single subroutine?

There is a lot of repetition in the data. Especially in the second and third voices. Can you find a way to redefine the data, so that a pattern that is repeated need only be entered once?

9

Advanced Music Techniques

In this chapter, we will examine some additional features of the SID chip and programming techniques that will add a degree of refinement to your sound. First the use of filters will be applied to our programs, then resonance. These features can also be used when producing sound effects to be discussed in the next chapter.

FILTERS

Earlier in this section on music, the concept of sound frequency and harmonics was explained. All sound consists of a fundamental frequency and certain harmonics depending on the waveform used.

The SID chip comes equipped with three kinds of filters that let you accept some of the frequencies that are produced and reject others. Frequencies that are passed are called resonated frequencies which increase the amplitude of the frequency. Frequencies that are rejected by a filter are attenuated frequencies, which decrease the amplitude. The three filters are called the low-pass, high-pass, and bandpass filters.

Low-Pass Filter

The low-pass filter may be used to accept frequencies below a set level. To use this or any other filter, a cutoff frequency is chosen. With the low-pass filter, frequencies below the cutoff are passed, but those harmonics above the cutoff are progressively attenuated. This tends to mute the sound since the more colorful higher harmonics are missing.

High-Pass Filter

With the high-pass filter, harmonics above the cutoff frequency will pass through unchanged, but frequencies below this point are attenuated. Since the fundamental frequency is the lowest, it will be the one with the greatest decrease in amplitude, leaving mainly harmonics. This filter is not very useful for music effects, since the fundamental frequency is missing.

Bandpass Filter

The bandpass filter passes a middle range of frequencies by attenuating both the low and high frequencies. Those frequencies that surround the cutoff frequency will be allowed to pass through the filter. Frequencies that are higher or lower than the cutoff will be progressively attenuated.

Band Reject Effect

The effect of a fourth filter is possible by combining a high- and low-pass filter. Using both of these filters together has an effect which is the opposite of a bandpass filter. In other words, frequencies at both the low and high ends are allowed to pass, but frequencies in the middle of the band are rejected.

USING THE FILTERS

There are several steps required to use a SID filter. These steps may be summarized as follows:

Step	*Description*	*Register*	*Address*
1	Select the voice(s) to filter	S+23	54295
2	Set the filter mode	S+24	54296
3	Choose the cutoff frequency	S+21	54293
		S+22	54294
4	Add resonance if desired	S+23	54295

SELECTING THE VOICE TO FILTER

Register S+23 is used to select the voice or voices you want to filter. Bit values 1, 2, and 4 of this register select voices 1, 2, and 3 for filtering. Bit value 8 is for filter X which is used for external audio inputs. Later we will see how to use the remaining four bits to add resonance to the sound.

S=54272

Voice to Filter S+23

128	64	32	16	8	4	2	1	Decimal value
0	0	0	0	0	0	0	1	
RES 3	RES 2	RES 1	RES 0	FILTEX	FILT 3	FILT 2	FILT 1	

To use a filter with voice 1, the following poke would be used.

```
100 POKE S+23,1
```

If both voice 1 and voice 3 are to be filtered use

```
100 POKE S+23,5
```

since value 4 represents voice 3 and a 1 represents voice 1. Adding 4 and 1 together gives the poke value of 5.

SETTING THE FILTER MODE

Register S+24 is used to select the type of filter; low-pass, high-pass or bandpass. Since this register is also used for the volume setting, these two values must be combined to provide for a composite setting.

S=54272

Set Filter Type S+24

128	64	32	16	8	4	2	1	Decimal value
0	0	0	1	1	1	1	1	
3 OFF	HIGH	BAND	LOW	VOL 3	VOL 2	VOL 1	VOL 0	
Filter Mode				Volume Setting				

To set the filter type to a low-pass filter with a maximum volume setting as shown in the diagram, the following values would be used:

Low-Pass Filter	16
Maximum Volume	15
Sum	31

This setting would be

```
110 POKE S+24,31
```

A high-pass filter with maximum volume would be set as follows:

High-Pass Filter	64
Maximum Volume	15
Sum	79

The poke would be

```
110 POKE S+24,79
```

The high order bit is unused for these filter settings. Its purpose is to turn voice 3 off when its frequency setting is used to modulate other voices.

CHOOSING THE CUTOFF FREQUENCY

The cutoff setting uses a 16-bit register by combining the values of registers S+21 and S+22. Register S+21 provides the three low order bit positions and register S+22 the eight high order bits.

S=54272

Cutoff Frequency

Low Order Bits S+21

128	64	32	16	8	4	2	1	Decimal value
0	0	0	0	0	0	0	1	
					FC2	FC1	FC0	

High Order Bits S+22

1	0	0	0	0	0	0	0
FC10	FC9	FC8	FC7	FC6	FC5	FC4	FC3

To set the cutoff as shown in the above chart the following pokes would be used:

```
120 POKE S+21,1:POKE S+22,128
```

This setting will require some experimentation to get the setting that works best for specific sounds. Try changing register S+22 first for the greatest effect, then register S+21 for subtle variations. Interesting effects can also be created by varying the values of these registers as the music is playing.

ADDING RESONANCE

Additional resonance may be added to the sound to increase the brightness of the voice. Since resonance is part of register S+23, its value will need to be added to the voice setting to give a composite poke setting.

Minimum resonance is achieved with a 0 value, and a maximum effect will be produced with a 240 setting.

S=54272

Resonance and Voice to Filter S+23

128	64	32	16	8	4	2	1	Decimal value
0	1	1	0	0	0	0	1	
RES 3	RES 2	RES 1	RES 0	FILTEX	FILT 3	FILT 2	FILT 1	

To set resonance to a medium value and apply a filter to voice 1, as shown in the above diagram, the following values and poke can be used:

Resonance Setting	96
Voice to Filter	1
Sum	97

```
100 POKE S+23,97
```

"WILLIAM TELL" WITH FILTERING

The following program is a revised "William Tell" from Chapter 7. The program now uses the low-pass filter and a resonance setting of 128 plus 1 for the voice selection.

```
10 REM WILLIAM TELL WITH FILTERING
20 AD= 17:SR=130:WF=33
30 FOR C=54272 TO 54296:POKE C,0:NEXT
40 S=54272
```

```
45 POKE S+21,1:POKE S+22,128
50 POKE S+23,129:POKE S+24,31
60 POKE S+5,AD:POKE S+6,SR
70 READ N1,N2,D
80 IF N1=-1 THEN POKE S+24,0:STOP
90 POKE S+1,N1:POKE S+0,N2
100 POKE S+4,WF
110 FOR I=1 TO D*100:NEXT
120 POKE S+4,WF-1
130 GOTO 70
1000 DATA 37,162,2,0,0,2,37,162,1,37,162,1,37,162
1005 DATA 4,0,0,2,37,162,1,37,162,1
1010 DATA 37,162,1,37,162,1,37,162,2,31,165,2,25,30,231,
    165,2
1020 DATA 31,165,2,37,162,2,31,165,2,37,162,2,50,60,2
1025 DATA 37,162,2,31,165,2,25,30,2
1030 DATA 31,165,2,37,162,2,31,165,2,37,162,2,47,107,2
1040 DATA 37,162,2,37,162,1,37,162,1,37,162,2,37,162,
    137,162,1
1050 DATA 37,162,1,37,162,2,37,162,2,37,162,2,-1,-1,-1
```

Comments on the Program

Other than the additional pokes needed to set the filter and resonance, the program is unchanged from the original.

Lines	
45	Sets the low order bits of the cutoff register to 1 and the high order bits to 128.
50	Sets resonance to 128 and voice 1 to be filtered for a poke value of 129 in register S+23.
	Sets maximum volume 15 and a low-pass filter 16 for a poke value of 31 for register S+24.

Challenge

Experiment with different cutoff frequencies with the low-pass filter. Try other resonance settings too. Now try using other filters and observe the effect. Try using some of these settings in other programs where the ADSR is set differently from this program. What is the effect in these programs?

SIMPLIFYING MUSIC DATA ENTRY

By now you may be finding the preparation and entry of the music rather tiresome. It does take some time and effort to take a piece of music, write down all of the decimal frequency values and durations for each note; then do this again for up to three parts of the music.

The program that follows cuts out one step in this process. Obviously it is necessary to tell the computer what the notes should be and the timing of each note. But why can't we get the computer to convert the notes themselves into the appropriate decimal values? With a table of values, this should be possible.

At first glance it might seem that a table of values for all 8 octaves would be needed. With the sharps included that would mean a table of 96 entries for the decimal values themselves. An additional 96 values would be needed to relate these values to the notes and octave.

But if you recall that each octave is double the decimal value of the previous octave, then the table can be simplified. If a note's decimal value is doubled to go up an octave, it must be halved to go down an octave. This is our clue.

Table 9.1 shows values for the 7th octave. We can move down to lower octaves by simply dividing the value by 2 several times until the octave we need is reached. The table will look as follows:

Table 9.1 Values for the 7th Octave

DIM N$(12),X(12)	
C	34334
C#	36376
D♯	38539
D♯	40830
E	43258
F	45830
F♯	48556
G	51443
G♯	54502
A	57743
A♯	61176
B	64814

With a table such as this, we will be able to record the music with notes such as

```
A    A#    B    D    F#
```

and so on. In addition to the notes, the program will need to know which octave the note is in. So an additional value will be attached to each note to indicate the octave. Thus a C in the 3rd octave will be coded as

```
C3
```

and an F# in the 6th octave will be recorded as

```
F#6        or        F6#
```

In addition to the note and octave, a duration value is needed. This value will be entered as in previous programs with a separate numeric value. A 3rd octave, C quarter note is entered as

```
1010 DATA C3,2
```

since a quarter note has a duration value of 2. A half note A# in the 5th octave is entered after the previous note as

```
1010 DATA C3,2,A#5,4
```

Since much of the melody is often in the 4th octave, the program is written to assume that a note written without an octave number will be in octave 4. For example,

```
1020 DATA B,2
```

is assumed to be a B quarter note in octave 4.

"THE ENTERTAINER"

In the following program, "The Entertainer" is played in three parts using the data entry method described above. In this program the first set of data contains values for the table. These numbers are followed by data for the first voice, which ends with -1 values, then the second voice, and, finally, the third voice.

```
100 REM THE ENTERTAINER
110 DIM H%(100,2),L%(100,2),D%(100,2)
115 DIM N$(12),X(12)
120 AD(0)=18:SR(0)=138:WF(0)=33
130 AD(1)= 9:SR(1)= 68:WF(1)=17
140 AD(2)=17:SR(2)=130:WF(2)=33
150 FOR C=54272 TO 54296:POKE C,0:NEXT
160 S=54272
170 I=0:V=0
175 GOSUB 800
180 GOSUB 300
190 POKE S+24,15
200 POKE S+5,AD(0):POKE S+6,SR(0)
210 POKE S+12,AD(1):POKE S+13,SR(1)
220 POKE S+19,AD(2):POKE S+20,SR(2)
230 I=0
240 GOSUB 400
250 POKE S+24,0
260 STOP
300 REM READ MUSIC DATA
305 PRINT CHR$(147); TAB(15); "W A I T"
310 READ F$,T
320 IF T=-2 THEN RETURN
330 IF T=-1 THEN N(V)=I-1:I=0:V=V+1:GOTO 310
335 GOSUB 900
340 H%(I,V)=F/256
350 L%(I,V)=F-(256*H%(I,V))
360 D%(I,V)=T
370 I=I+1
380 GOTO 310
400 REM PLAY MUSIC
410 GOSUB 500
420 GOSUB 600
430 GOSUB 700
440 FOR D=1 TO 80:NEXT
450 D1=D1-1:D2=D2-1:D3=D3-1
460 IF D1=0 THEN POKE S+4,WF(0)-1:GOSUB 500
470 IF D2=0 THEN POKE S+11,WF(1)-1:GOSUB 600
480 IF D3=0 THEN POKE S+18,WF(2)-1:GOSUB 700
490 IF N1=N(0) AND N2=N(1) AND N3=N(2) THEN RETURN
495 GOTO 440
500 REM VOICE 1
510 IF N1=N(0) THEN RETURN
520 POKE S+1,H%(N1,0):POKE S+0,L%(N1,0)
530 D1=D%(N1,0)
540 POKE S+4,WF(0)
550 N1=N1+1
560 RETURN
```

```
600 REM VOICE 2
610 IF N2=N(1) THEN RETURN
620 POKE S+8,H%(N2,1):POKE S+7,L%(N2,1)
630 D2=D%(N2,1)
640 POKE S+11,WF(1)
650 N2=N2+1
660 RETURN
700 REM VOICE 3
710 IF N3=N(2) THEN RETURN
720 POKE S+15,H%(N3,2):POKE S+14,L%(N3,2)
730 D3=D%(N3,2)
740 POKE S+18,WF(2)
750 N3=N3+1
760 RETURN
800 REM LOAD BASE FREQUENCY
810 FOR K=1 TO 12
820 READ N$(K),X(K)
830 NEXT
840 RETURN
900 REM CONVERT LETTERS TO FREQUENCY
902 O=4:L$=""
903 IF F$="0" THE F=0:RETURN
905 FOR K=1 TO LEN(F$)
910 C$=MID$(F$,K,1)
915 IF C$>="A" AND C$<="G" THEN L$=C$
920 IF C$<="0" AND C$<="9" THEN O=VAL(C$)
925 IF C$="#" THEN L$=L$+C$
930 NEXT
935 FOR K=1 TO 12
940 IF N$(K)=L$ THEN F=X(K):GOTO 950
945 NEXT
946 IF O=7 THEN RETURN
950 FOR J=O+1 TO 7
955 F=F/2
960 NEXT
990 RETURN
1000 DATA C,34334,C#,36376,D,38539,D#,40830,E,43258,F,45830
1001 DATA F#,48556,G,51443,G#,54502,A,57743,A#,61176,
     B,64814
1010 DATA E,1,C5,2,E,1,C5,2,E,1,C5,1
1020 DATA C5,4,C5,1,C5,1,D5,1,D#5,1
1030 DATA E5,1,C5,1,D5,1,E5,1,E5,1,B4,1,D5,2
1040 DATA C5,6,D,1,D#,1
1050 DATA E,1,C5,2,E,1,C5,2,E,1,C5,1
1060 DATA C5,6,A,1,A,1
1070 DATA F#,1,A,1,C5,1,E5,1,E5,1,D5,1,C5,1,A,1
1080 DATA B,6,0,2,-1,-1
1100 DATA C3,2,G3,2,G3,2,A#3,2
1110 DATA F3,2,A3,2,E3,2,G3,2
1120 DATA G3,2,C,2,G3,2,B3,2
1130 DATA C,2,G3,2,G3,2,0,2
1140 DATA C3,2,G3,2,G3,2,A#3,2
1150 DATA F3,2,A3,2,E3,2,D#3,2
1160 DATA D3,2,F#3,2,D3,2,F3,2
1170 DATA G3,2,G2,2,A2,2,B2,2,-1,-1
```

```
1200 DATA 0,2,C,2,0,2,C4,2
1210 DATA 0,2,C,2,0,2,C,2
1220 DATA 0,2,E,2,0,2,F,2
1230 DATA E,2,0,6
1240 DATA 0,2,C4,2,0,2,C4,2
1250 DATA 0,2,C4,2,0,4
1260 DATA 0,2,C4,2,0,2,C4,2,0,8,-1,-1,-2,-2
```

Comments on the Program

Since much of this program is based on the "Skaters Waltz" from Chapter 8, only the revised sections of code will be explained here.

Lines	
115	These tables hold the notes (N$) and decimal values (X) for the 7th octave frequencies.
175	Subroutine 800 loads the frequency tables.
310	The READ now reads a string for each note, such as C#5, and a duration value (T).
335	Subroutine 900 converts the note string to a decimal frequency value in the correct octave.
800–840	Load the tables with 12 notes and frequency values for octave 7.
900–960	Convert each note string to the appropriate decimal value in F for the required octave.
902	Initializes octave (O) to zero and L$ to an empty string.
903	A zero value represents a rest. Simply return zero in variable F.
905–930	Loop to extract parts of the note string.
910	Extracts one character.
915	If the character is a letter note just assign it to L$.
920	If the character is an octave number converts the number to a numeric value and stores it in O.
925	A sharp is concatenated to L$.
935–945	Find the location of the note in the table and assign its equivalent decimal value to F.
946	Octave 7 notes do not need to be adjusted so return from the subroutine.
950–960	Divide the frequency by 2 for each octave. For example, to get from octave 7 to octave 5, the frequency value will be divided by 2 twice. To get to octave 3, F would be divided 4 times.
1000–1001	Data for the frequency tables.
1010–1080	Voice 1 data.
1100–1170	Voice 2 data.
1200–1260	Voice 3 data.

Challenge

Add some filtering to this program. Try experimenting with filters on one or more of the voices. You may find that filtering one voice will cause another to stand out. Try adjusting the ADSR settings and the waveform for additional control over the effects this music will produce. There is a wide range of possible sound that can be produced, so experiment to your heart's content.

10

Sound Effects

In addition to music, the Commodore 64's SID chip also has the capability for producing great sound effects. These effects are lots of fun and may also be combined with the graphics in the first half of the book to create super results.

Sound effects can be produced in several ways. One is to use voices 1 to 3 and control the frequency of the voice(s) to produce effects like sirens, lasers, and space sounds. We will use this technique first in this chapter. A second method is to use the white noise voice. This voice can produce some very interesting effects like explosions or drums. The third technique is to combine white noise with another voice.

THE BASIC SOUND EFFECT TECHNIQUE

Basic sound effects can be created by using one of the voices we used earlier for music. Several voices can also be used for more complex effects. A major difference between music and sound effects is that the latter does not usually require specific notes to be produced. So the following program will vary only the high frequency register.

The technique used is to vary a value with a FOR loop and use this value to poke the high frequency register. A loop such as

```
170 FOR I=0 TO 255 STEP 1
180 POKE S+1,I
200 NEXT I
```

creates values from 0 to 255 in increments of one in variable I. This number is poked into the high frequency register (S+1) each time through the loop.

In the music programs, we found the need for a delay loop, otherwise the sound changed too rapidly. Although rapid changes are often necessary for sound effects, here too we will need to delay changes at times to get the desired effect. So a delay loop is often used within the sound loop.

```
170 FOR I=0 TO 255 STEP 1
180 POKE S+1,I
190 FOR D=1 TO 30:NEXT
200 NEXT I
```

The ADSR for sound effects is often not as critical as for music, but normally we want a fast attack, no decay, a long sustain, and a fast release. As far as the waveform is concerned, once it is set, we will not usually need to turn it off.

Now try running this basic sound effect program.

```
100 REM BASIC SOUND EFFECT
110 FOR C=54272 TO 54296:POKE C,0:NEXT
120 S=54272
130 POKE S+24,15
140 POKE S+5,240
150 POKE S+6,240
160 POKE S+4,33
170 FOR I=0 TO 255 STEP 1
180 POKE S+1,I
190 FOR D=1 TO 30:NEXT
200 NEXT I
210 POKE S+24,0
```

Comments on the Program

Lines	
110–160	Initialize SID registers, set volume, ADSR and waveform.
170–200	Loop to provide a high frequency poke value. Low frequency values are unnecessary and will have little effect on the sound if used.
180	Pokes the high frequency register.
190	Time delay loop.
210	Turns off volume. Waveform could also be turned off here but that is not essential.

Challenge

Try varying the values in statement number 170. Start with changing the step value as follows:

```
170 FOR I=0 TO 255 STEP 3
```

Try a few other step values and observe the effect. Now try adjusting the start and end values. Maybe start at 50 and go up to 200. Try some other variations.

SOUND VARIATIONS

The above challenge has brought your sound close to the effects produced in the next program. In it, a narrower range of values is used to poke the high frequency register. Also a step value of 3 is used. Both of these changes give the effect of sliding the tone from its low to its high note very quickly. To further enhance this smooth, fast rise in tone, the delay loop is a short 10 repetitions.

With all of these changes, the sound effect would not last very long, only a fraction of a second. So this entire sound loop is placed within a second loop which repeats the sound effect 20 times. Using repetition gives an entirely new quality to the sound.

```
100 REM SOUND VARIATIONS
110 FOR C=54272 TO 54296:POKE C,0:NEXT
120 S=54272
130 POKE S+24,15
140 POKE S+5,240
150 POKE S+6,240
160 POKE S+4,33
165 FOR K=1 TO 20 STEP 1
170 FOR I=30 TO 70 STEP 3
```

```
180 POKE S+1,I
190 FOR D=1 TO 10:NEXT
200 NEXT I:NEXT K
210 POKE S+24,0
```

Comments on the Program

Lines	
100–160	These statements are unchanged from the basic sound program.
165–200	Loop K controls the number of times the sound effect is repeated.
170–200	Loop I controls the range of the tone from low to high and the step value affects how quickly it rises to the high level.

Challenge

Vary the values in statement 170 and in the delay loop. What changes in the sound effect can you produce?

USING REVERSE LOOPS

So far all of the tone generating loops have used positive increments to produce sound ranging from a low to a high tone. What happens if we reverse this and use a negative increment? Here is the same program with statement 170 going from 70 to 30 in steps of -3. Notice the difference.

```
100 REM SOUND VARIATIONS
110 FOR C=54272 TO 54296:POKE C,0:NEXT
120 S=54272
130 POKE S+24,15
140 POKE S+5,240
150 POKE S+6,240
160 POKE S+4,33
165 FOR K=1 TO 20 STEP 1
170 FOR I=70 TO 30 STEP -3
180 POKE S+1,I
190 FOR D=1 TO 10:NEXT
200 NEXT I:NEXT K
210 POKE S+24,0
```

USING TWO SOUND LOOPS

If a single sound loop varying the tone from either low to high or the reverse is an effective technique, what would happen if we used both of these types of loops in one program? One loop would take the sound from a low to a high tone, and the other would then vary it from the high value back to the low. This would require two loops following a pattern something like the following:

```
170 FOR I=30 TO 70 STEP 3
  .                                      loop to vary sound
  .                                      from low to high tone
  .
200 NEXT I
210 FOR I=70 TO 30 STEP -3
  .                                      loop to vary sound
  .                                      from high to low tone
  .
240 NEXT I
```

When creating most sound effects, you will need to experiment with different values to get the effect you want. So even though the values of 30 and 70 in the loops worked well for the previous program, they may not be what you want here. So feel free to try any values in the range of 0 to 255.

Another interesting technique is to vary the starting and ending values of these loops as the sound is repeated. The method is to use the variable from the repetition loop to change the values in the sound loops. Here is the structure:

```
165 FOR K=1 TO 30
170 FOR I=30-K TO 70-K STEP 3
 .                                          loop to vary sound
 .                                          from low to high tone
 .
200 NEXT I
210 FOR I=70-K TO 30-K STEP -3
 .                                          loop to vary sound
 .                                          from high to low tone
 .
240 NEXT I
```

Reducing each value by K has the effect of starting and ending the tone slightly lower each time through the loop. Of course, this adjustment could be applied to just one of the loops for a different effect. It could also be applied to the starting value but not to the ending value or vice versa.

Here is the revised program that uses these techniques. The values have again been changed slightly to produce a new effect. It sounds something like the Martians landing in their UFO.

MARTIANS

```
100 REM MARTIANS
110 FOR C=54272 TO 54296:POKE C,0:NEXT
120 S=54272
130 POKE S+24,15
140 POKE S+5,240
150 POKE S+6,240
160 POKE S+4,33
165 FOR K=1 TO 40 STEP 1
170 FOR I=50-K TO 90-K STEP 5
180 POKE S+1,I
190 FOR D=1 TO 5:NEXT
200 NEXT I
210 FOR I=90-K TO 50-K STEP 5
220 POKE S+1,I
230 FOR D=1 TO 10:NEXT
240 NEXT I:NEXT K
250 POKE S+24,0
```

Comments on the Program

Lines	
165–240	Loop K controls the number of repetitions of the sound effect and provides an adjustment value for the starting and ending tones.
170–200	Loop creates the low to high tone adjusted each time by the value of K.
210–240	This loop produces the high to low tone also adjusted by the value of K each time.

Challenge

What happens if you use different values and ranges for loops 170 and 210? Try also adjusting the starting value with K but not the ending value. Make one loop's range greater than the other loop. You can create all kinds of crazy results by varying these values. Have fun!

USING ALTERNATING TONES

Switching between two or more tones can produce interesting sound effects. One of the more popular of these effects is the high-low alternating tone that sounds like a siren.

Like most sound effects you will need to do some experimentation to get the exact tones you want for a particular effect. In this program a high tone (frequency 80) is poked into the high frequency register. The program then waits for a time delay before poking the low value of 40. Following the low tone is another time delay of equal length to the previous one.

SIREN

```
100 REM SIREN
110 FOR C=54272 TO 54296:POKE C,0:NEXT
120 S=54272
130 POKE S+24,15
140 POKE S+5,240
150 POKE S+6,240
160 POKE S+4,33
170 FOR K=1 TO 30
180 POKE S+1,80
190 FOR D=1 TO 300:NEXT
220 POKE S+1,40
230 FOR D=1 TO 300:NEXT
240 NEXT K
250 POKE S+24,0
```

Comments on the Program

Lines	
140	Sets slow attack and slow decay rates.
150	Sets long sustain and fast release. This setting is not critical since the waveform is never turned off.
160	Sawtooth waveform.
170–240	Loop to repeat the high-low sequence 30 times.
180	Plays the high note.
190	Time delay while high note is held constant.
220	Plays the low note.
230	Holds the low note for 300 repetitions.
250	Turns off the sound.

Challenge

What other sounds consist of an alternating high and low frequency? Try creating a few of them. What happens if you use two voices to produce each of these tones? Try it with a different tone for each voice to produce even more harmonics.

GENERATING RANDOM SOUNDS

Some sounds do not occur with any regularity. Thunder, rain, insect noises, birds, and so on have a seemingly random property to them. Random numbers can be generated by the Commodore 64 using the RND function and can be used in several ways when producing sound.

One use of random numbers is to produce a variety of frequencies of sound. In this case, the RND function creates a value which is used to poke the high frequency register. The function can also control the range of values, so that the tones generated are within a predefined range.

A second use of random numbers is to control the time delay between sounds. This use of RND will generate a value to be used by the time delay loop. Again a specific range of values to work with is usually necessary.

The RND function is used as follows:

```
N=INT(RND(1)*10+1)
```

This statement will generate random numbers in the range of 1 to 10. A statement such as

```
N=INT(RND(1)*255+1)
```

would generate values in the range of 1 to 255. The value in N can then be used in other statements such as a poke or delay loop.

The following program I've called Computer Talk. It uses the RND function to produce random sounds.

COMPUTER TALK

```
100 REM COMPUTER TALK
110 FOR C=54272 TO 54296:POKE C,0:NEXT
120 S=54272
130 POKE S+24,15
140 POKE S+5,240
150 POKE S+6,240
160 POKE S+4,33
170 FOR K=1 TO 200
180 N=INT(RND(1)*80+20)
190 POKE S+1,N
200 FOR D=1 TO 40:NEXT
210 NEXT K
220 POKE S+24,0
```

Comments on the Program

Lines	
110–160	Set SID registers.
170–210	Do 200 repetitions of these sounds.
180	Generates random numbers in the range of 20 to 100.
190	Pokes the high frequency register.
200	Time delay loop.

Challenge

Create some variations on the theme by using other ranges of the random number. Then use a random number generator in the delay loop in statement 200, so that the time between each sound is varied.

WHITE NOISE

The last type of sound that is available in the Commodore 64 is the white noise generator. This sound is available from all voices by using a waveform setting of 129. Noise is turned off by poking the waveform register with a 128 value.

You recall that white noise is composed of many frequencies and harmonics, whereas other waveforms have very distinctive patterns. But, even so, white noise is affected by the attack/decay, sustain/release, and frequency settings.

To demonstrate the use of white noise the following program simulates the sound of a steam engine moving off from the station gaining speed as it goes.

STEAM ENGINE

```
100 REM STEAM ENGINE
110 FOR C=54272 TO 54296:POKE C,0:NEXT
120 S=54272
130 POKE S+24,15
140 POKE S+5,128:POKE S+6,240
150 J=400
160 FOR K=1 TO 20
170 POKE S+4,129
180 POKE S+1,80
190 FOR D=1 TO 30:NEXT
200 POKE S+4,128
210 FOR D=1 TO J:NEXT
220 J=J-3
230 NEXT K
240 POKE S+24,0
```

Comments on the Program

Lines	
110–130	Initialize register settings.
140	Slow attack and no decay. Longest sustain and fastest release.
150	J controls the increase in speed of the sound.
160–230	Loop to repeat the sound of the train.
170	Sets the noise waveform.
180	Sets the frequency.
190	Constant time delay for the sound's duration.
200	Turns off the waveform to stop the sound temporarily.
210	Controls the time delay between chugs. In other words, this loop creates the effect of how fast the train is moving.
220	Reduces J's value so the train will chug slightly faster each time through the K loop.

Challenge

Now that you see how the noise waveform is used, go back to some of the previous programs and try substituting the noise waveform for the other waveform settings that had been used. Try also adjusting the ADSR settings.

USING THE RANDOM FUNCTION WITH WHITE NOISE

Like other sounds the RND function can be used to generate random white noise sounds. By using appropriate ADSR and frequency settings, interesting and unusual sounds can be created. The importance of the ADSR and frequency settings should be reemphasized here since these settings will make all the difference in the effects that you get.

In the next program, the settings are made to simulate gunfire. The RND function is used to control the time delay between shots so that the pattern seems more realistic.

GUNFIRE

```
100 REM GUNFIRE
110 FOR C=54272 TO 54296:POKE C,0:NEXT
120 S=54272
130 POKE S+24,15
140 POKE S+5,0:POKE S+6,250
150 FOR K=1 TO 20
160 POKE S+4,129
170 POKE S+1,20
180 FOR D=1 TO 40:NEXT
190 POKE S+4,128
200 T=INT(RND(1)*500+100)
210 FOR D=1 TO T:NEXT
230 NEXT K
240 POKE S+24,0
```

Comments on the Program

Lines	
110–130	Initialize SID registers.
140	Fast attack, no decay, maximum sustain, and slow release simulates a single shot of gunfire.
150–230	Shoots 20 rounds of ammunition.
160	Sets the noise waveform.
170	Frequency setting.
180	Time delay for the duration of the sound of one shot.
190	Turns off the waveform.
200	Calculates the time value to delay between shots.
210	Time delay between shots.

Challenge

Change the frequency setting to different values and notice the effect this has on the sound. Changing the ADSR settings will also affect the quality of the sound, in some cases so much so, that it will no longer sound like gunfire.

COMBINING SOUND EFFECTS

Some sounds will be composed of two or more separate sounds. In the next program, we will use two different waveforms and ADSR settings to create the sound of a bomb falling and the explosion.

The first part of the program produces the whistling sound of the bomb falling toward the earth, by using a sawtooth waveform and a declining frequency setting. This is a separate loop from the explosion which is to follow. In this case, you can write and test this part of the program without the need to have solved the second part.

The second half of the program creates the explosion. This uses the white noise waveform and a constant frequency setting.

Don't make the mistake of thinking that this program was simply written and that it worked the first time. In reality, this program and others that have preceded it in this chapter required some experimentation with the settings to get the effect I wanted. Your programs will no doubt also require experimenting with different settings to get the effects you want. Don't give up; keep on trying until you succeed.

BOMB

```
100 REM BOMB
110 FOR C=54272 TO 54296:POKE C,0:NEXT
120 S=54272
130 POKE S+24,15
140 POKE S+5,240
150 POKE S+6,240
160 POKE S+4,33
170 FOR K=230 TO 60 STEP -2
180 POKE S+1,K
190 FOR D=1 TO 30:NEXT
200 NEXT K
210 POKE S+4,32
220 POKE S+5,4
230 POKE S+6,251
240 POKE S+4,129
250 POKE S+1,3
260 FOR D=1 TO 200:NEXT
270 POKE S+4,128
280 FOR D=1 TO 2500:NEXT
290 POKE S+24,0
```

Comments on the Program

Lines	
110–130	Initialize SID settings.
140–210	Whistling sound of the falling bomb.
140	Attack/decay setting.
150	Sustain/release setting.
160	Sawtooth waveform setting.
170–200	Frequency control loop.
180	Pokes the frequency value.
190	Delay loop.
210	Turns off the waveform to end the whistle.
220–290	Create the explosion.
220	New attack/decay setting.
230	New sustain/release.
240	Sets the white noise waveform.
250	Pokes the frequency of the explosion. Notice the low value of this setting.
260	Time delay of the main part of the explosion.
270	Turns off the noise waveform. But since a very slow release setting is used, the noise will continue for some time.
280	Waits for awhile to let the sound of the explosion fade out. This loop is needed to take advantage of the slow release.
290	Turns off the volume.

Challenge

What other sounds consist of two or more single sounds combined? Try to produce some of these. Choose simple effects at first and then as you master these, move on to more complex ones.

11

The VIC and SID Show

Sound effects by themselves are fun and interesting, but combine them with an animated graphic and they become even more impressive. In this book, we have seen how to create and animate sprite graphics. We have also created sound effects. Now in this last chapter, you will learn how to combine the power of the VIC-II chip to create a sprite and the power of the SID chip to create sound into one effective program.

COMBINING GRAPHICS AND SOUND

For this last program, we will take the chopper landing program from chapter 5 and combine this sprite with the sound of the chopper blades whipping around in the wind as the chopper lands on the Tarmac.

When developing a program of this type, it is usually advisable to develop the program code to land the helicopter and make sure it works, then do the same with the sound effects without the chopper. When both parts of the program are satisfactory, then you can combine them into one program.

The technique used when combining the two parts is a GOSUB in the sprite's animation loop. Every time the sprite is moved one position, the sound subroutine is called and one set of sound is produced. Then the program returns, moves the sprite one more position, and goes back to produce sound once again. This happens fast enough so that the movement and the sound seem to occur together.

This step of combining the two parts will often have problems, since there will be timing interactions between the two parts. Since the chopper takes some time to move, this will have the effect of slowing down the sound. Also, since there are delay loops used in the sound effects, these loops will slow down the landing of the chopper.

The solution? Well, one solution is to increase the speed of the sprite by using larger increments in its loop. Of course, too large an increment will result in jerky movement. Another solution is to reduce the time spent in the sound delay loop. This is a good solution because some of the delay can come from the time the chopper takes in its movement.

In practice you will need to fine tune the program at this stage for the best effect.

VIC AND SID

```
100 REM VIC AND SID
110 PRINT CHR$(147)
120 V=53248:S=54272
130 POKE 2040,192
```

```
140 FOR D=0 TO 62:READ N:POKE 192*64+D,N:NEXT
150 POKE V+21,1
160 FOR C=54272 TO 54296:POKE C,0:NEXT
170 POKE S+24,15
180 POKE S+5,140:POKE S+6,4
190 FOR Y=50 TO 230 STEP 2
200 POKE V+0,255-Y:POKE V+1,Y
210 GOSUB 500
220 NEXT Y
230 POKE S+24,0
240 STOP
500 REM SOUND
510 FOR K=1 TO 5
520 POKE S+4,129
530 POKE S+1,20
540 FOR D=1 TO 20:NEXT
550 POKE S+4,128
560 FOR D=1 TO 30:NEXT
570 NEXT K
580 RETURN
1000 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,255,255,255
1010 DATA 0,48,0,0,48,0,0,48,0,65,48,0,65,240,0
1020 DATA 128,120,1,129,7,255,135,23,1,127,240,0
1030 DATA 8,128,0,8,128,0,127,255,128,0,0,0,0,0,0,0,0
```

Comments on the Program

Lines	
110–150	Initialize the sprite data and registers.
160–180	Initialize SID registers.
190–220	Loop to animate sprite.
200	Positions sprite at the required X and Y coordinates.
210	Go to the sound subroutine.
230–240	Turn off the sound and stop the program when the chopper has landed.
500–580	Sound subroutine.
510–570	Loop to repeat sound 5 times.
520	Sets white noise waveform.
530	Sets frequency.
540	Holds sound for a delay of 20 loops.
550	Turns off waveform.
560	Gives sound time to be released.
580	Returns to sprite display loop.
1000–1030	Sprite data.

Challenge

Now use your imagination to see what graphic and sound effects you can produce. You have now mastered the principles of graphics, sound, and music. The rest is up to your imagination. Happy programming!

Appendix A

Commodore 64 Color Codes and Combinations

Table A.1 Border and Background Colors

Border color	POKE 53280,X
Background color	POKE 53281,Y

Values for X and Y

Black	0	Orange	8
White	1	Brown	9
Red	2	Light Red	10
Cyan	3	Dark Gray	11
Purple	4	Medium Gray	12
Green	5	Light Green	13
Blue	6	Light Blue	14
Yellow	7	Light Gray	15

Table A.2 Excellent Color Combinations

		CHARACTER COLOR															
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	0		X		X	X			X	X		X	X	X	X	X	X
	1	X		X		X	X	X			X		X	X		X	X
S	2		X						X	X		X					
C	3	X						X									
R	4	X															
E	5	X													X		
E	6		X		X											X	X
N	7	X		X							X		X	X			
	8		X	X					X		X						
C	9		X						X	X		X					X
O	10			X							X						
L	11	X	X						X					X	X		X
O	12	X	X										X				X
R	13	X					X						X				
	14	X	X		X			X									
	15	X	X	X				X					X	X			

0	Black	4	Purple	8	Orange	12	Gray 2
1	White	5	Green	9	Brown	13	Light Green
2	Red	6	Blue	10	Light Red	14	Light Blue
3	Cyan	7	Yellow	11	Gray 1	15	Gray 3

Appendix B

Sprite Register Addresses

Sprite Registers

Memory

Addresses

2040	2041	2042	2043	2044	2045	2046	2047
0	1	2	3	4	5	6	7

Sprite Registers

Table B.1 Memory Allocation for Sprite Data

Sprite	0	1	2	3	4	5	6	7
Register	2040	2041	2042	2043	2044	2045	2046	2047
Memory Area	192	193	194	195	196	197	198	199
Starting Poke Address for Data	192*64	193*64	194*64	195*64	196*64	197*64	198*64	199*64

Sprite Enable/Disable Register V+21

V=53248

7	6	5	4	3	2	1	0	Sprite number
0	0	0	0	0	1	0	0	0—disable 1—enable
128	64	32	16	8	4	2	1	Decimal value

Table B.2 Sprite X and Y Coordinate Registers

Register	Sprite Coordinate
0	X Coordinate—Sprite 0
1	Y Coordinate—Sprite 0
2	X Coordinate—Sprite 1
3	Y Coordinate—Sprite 1
4	X Coordinate—Sprite 2
5	Y Coordinate—Sprite 2
6	X Coordinate—Sprite 3
7	Y Coordinate—Sprite 3
8	X Coordinate—Sprite 4
9	Y Coordinate—Sprite 4
10	X Coordinate—Sprite 5
11	Y Coordinate—Sprite 5
12	X Coordinate—Sprite 6
13	Y Coordinate—Sprite 6
14	X Coordinate—Sprite 7
15	Y Coordinate—Sprite 7
16	Most Significant Bit (MSB)

Most Significant Bit Register V+16

7	6	5	4	3	2	1	0	Sprite number
0	0	0	0	0	0	0	1	0—off 1—on
128	64	32	16	8	4	2	1	Decimal value

Table B.3 Sprite Color Registers and Codes

Register	Sprite
V+39	0
V+40	1
V+41	2
V+42	3
V+43	4
V+44	5
V+45	6
V+46	7

Table B.4 Color Codes

Color	Code	Color	Code
Black	0	Orange	8
White	1	Brown	9
Red	2	Light Red	10
Cyan	3	Dark Gray	11
Purple	4	Medium Gray	12
Green	5	Light Green	13
Blue	6	Light Blue	14
Yellow	7	Light Gray	15

Registers for Sprite Expansion

7	6	5	4	3	2	1	0	Sprite numbers 2 and 3 enabled
0	0	0	0	1	1	0	0	Expansion register V+23 Expand in Y direction
0	0	0	0	1	1	0	0	Expansion register V+29 Expand in X direction
128	64	32	16	8	4	2	1	Decimal value

Background Priority Register V+27

V=53248

7	6	5	4	3	2	1	0	Sprite number
0	0	0	0	0	1	0	0	0—sprite higher 1—sprite lower than background
128	64	32	16	8	4	2	1	Decimal value

Sprite-to-Sprite Collision Register V+30

V=53248

7	6	5	4	3	2	1	0	Sprite number
0	0	0	0	0	1	1	0	0—no collision 1—sprite collision
128	64	32	16	8	4	2	1	Decimal value

Sprite-to-Background Collision Register V+31

V=53248

7	6	5	4	3	2	1	0	Sprite number
0	0	0	0	0	1	0	0	0—no collision 1—sprite collision
128	64	32	16	8	4	2	1	Decimal value

Sprite Multicolor Register V+28

V=53248

7	6	5	4	3	2	1	0	Sprite number
0	0	0	0	0	1	0	0	0—single color sprite 1—multicolor sprite
128	64	32	16	8	4	2	1	Decimal value

Appendix C

Sound Registers

Table C.1 POKE Addresses for SID Sound Settings

S=54272 **Type of Register**	**Voice 1**		**Voice 2**		**Voice 3**	
	Actual Address	**Program Usage**	**Actual Address**	**Program Usage**	**Actual Address**	**Program Usage**
Low Frequency	54272	V+0	54279	V+7	54286	V+14
High Frequency	54273	V+1	54280	V+8	54287	V+15
Waveform	54276	V+4	54283	V+11	54290	V+18
Attack/Decay	54277	V+5	54284	V+12	54291	V+19
Sustain/Release	54278	V+6	54285	V+13	54292	V+20
Volume	54296	V+24	54296	V+24	54296	V+24

Table C.2 POKE Values for Octave Four Frequencies

S=54272

Voice Frequency POKE Values

Memory Addresses	**Voice 1**	**Voice 2**	**Voice 3**
High Frequency	S+1	S+8	S+15
Low Frequency	S+0	S+7	S+14

Notes in the Fourth Octave	**C**	**C#**	**D**	**D#**	**E**	**F**	**F#**	**G**	**G#**	**A**	**A#**	**B**	**C**	**C#**
High POKE	16	17	18	19	21	22	23	25	26	28	29	31	33	35
Low POKE	195	195	209	239	31	96	181	30	156	49	223	165	135	134
Decimal	4291	4547	4817	5103	5407	5728	6069	6430	6812	7217	7647	8101	8583	9094

Table C.3 Values for Attack/Decay Settings

S=54272

Attack/Decay POKE Values

Memory Addresses	**Voice 1**	**Voice 2**	**Voice 3**
Program Uses	S+5	S+12	S+19

	Attack Value	**Decay Value**	
Slow Attack	128	8	*Fast Decay*
	64	4	
	32	2	
	16	1	
Fast Attack	0	0	*Slow Decay*

Table C.4 Values for Sustain/Release Settings

S=54272

Sustain/Release POKE Values

Memory Addresses	**Voice 1**	**Voice 2**	**Voice 3**
Program Uses	S+6	S+13	S+20

	Sustain Value	**Release Value**	
Long Sustain	128	8	*Slow Release*
	64	4	
	32	2	
	16	1	
Short Sustain	0	0	*Fast Release*

Table C.5 Values for Waveform Settings

S=54272

Waveform POKE Values

Memory Addresses	**Voice 1**	**Voice 2**	**Voice 3**
Program Uses	S+4	S+11	S+18

	Start Value	**Stop Value**
Triangle	17	16
Sawtooth	33	32
Pulse	65	64
Noise	129	128

Table C.6 POKE Values for Pulse Rate

S=54272

Pulse Rate Values

Memory Addresses	**Voice 1**	**Voice 2**	**Voice 3**
High Pulse (0 to 15)	S+3	S+10	S+17
Low Pulse (0 to 255)	S+2	S+9	S+16

Voice to Filter S+23

S=54272

128	64	32	16	8	4	2	1	Decimal value
0	0	0	0	0	0	0	1	
RES 3	RES 2	RES 1	RES 0	FILTEX	FILT 3	FILT 2	FILT 1	

Set Filter Type S+24

S=54272

128	64	32	16	8	4	2	1	Decimal value
0	0	0	1	1	1	1	1	
3 OFF	HIGH	BAND	LOW	VOL 3	VOL 2	VOL 1	VOL 0	
	Filter Mode				Volume Setting			

Cutoff Frequency

S=54272

Low Order Bits S+21

128	64	32	16	8	4	2	1	Decimal value
0	0	0	0	0	0	0	1	
					FC2	FC1	FC0	

High Order Bits S+22

1	0	0	0	0	0	0	0
FC10	FC9	FC8	FC7	FC6	FC5	FC4	FC3

Resonance and Voice-to-Filter S+23

S=54272

128	64	32	16	8	4	2	1	Decimal value
0	1	1	0	0	0	0	1	
RES 3	RES 2	RES 1	RES 0	FILTEX	FILT 3	FILT 2	FILT 1	

Appendix D

Voice Frequency POKE Values

Table D.1 Voice Frequency POKE Values

Octave 0	C	C♯	D	D♯	E	F	F♯	G	G♯	A	A♯	B
High POKE	1	1	1	1	1	1	1	1	1	1	1	1
Low POKE	12	28	45	62	81	102	123	145	169	195	221	250
Decimal	268	284	301	318	337	358	379	401	425	451	477	506

Octave 1	C	C♯	D	D♯	E	F	F♯	G	G♯	A	A♯	B
High POKE	2	2	2	2	2	2	2	3	3	3	3	3
Low POKE	24	56	90	125	163	204	246	35	83	134	187	244
Decimal	536	568	602	637	675	716	758	803	851	902	955	1012

Octave 2	C	C♯	D	D♯	E	F	F♯	G	G♯	A	A♯	B
High POKE	4	4	4	4	5	5	5	6	6	7	7	7
Low POKE	48	112	180	251	71	152	237	71	167	12	119	233
Decimal	1072	1136	1204	1275	1351	1432	1517	1607	1703	1804	1911	2025

Octave 3	C	C♯	D	D♯	E	F	F♯	G	G♯	A	A♯	B
High POKE	8	8	9	9	10	11	11	12	13	14	14	15
Low POKE	97	225	104	247	143	48	218	143	78	24	239	210
Decimal	2145	2273	2408	2551	2703	2864	3034	3215	3406	3608	3823	4050

Octave 4	C	C♯	D	D♯	E	F	F♯	G	G♯	A	A♯	B
High POKE	16	17	18	19	21	22	23	25	26	28	29	31
Low POKE	195	195	209	239	31	96	181	30	156	49	223	165
Decimal	4291	4547	4817	5103	5407	5728	6069	6430	6812	7217	7647	8101

Octave 5	C	C♯	D	D♯	E	F	F♯	G	G♯	A	A♯	B
High POKE	33	35	37	39	42	44	47	50	53	56	59	63
Low POKE	135	134	162	223	62	193	107	60	57	99	190	75
Decimal	8583	9094	9634	10207	10814	11457	12139	12860	13625	14435	15294	16203

Octave 6	C	C♯	D	D♯	E	F	F♯	G	G♯	A	A♯	B
High POKE	67	71	75	79	84	89	94	100	106	112	119	126
Low POKE	15	12	69	191	125	131	214	121	115	199	124	151
Decimal	17167	18188	19269	20415	21629	22915	24278	25721	27251	28871	30588	32407

Octave 7	C	C♯	D	D♯	E	F	F♯	G	G♯	A	A♯	B
High POKE	134	142	150	159	168	179	189	200	212	225	238	253
Low POKE	30	24	139	126	250	6	172	243	230	143	248	46
Decimal	34334	36376	38539	40830	43258	45830	48556	51443	54502	57743	61176	64814

Index

"Amazing Grace," 122
"Amazing Grace with Bagpipes," 124
AND, 69
Animation, 31, 35, 38, 72
Arrays, 120
Attack and decay, 107

Background colors, 6
Background priority register, 90
Background to sprite priority, 87
Bar charts, 12
Bomb, 149
Boolean, 69
Border colors, 6
Bouncing ball, 43

Cars passing, 85, 87
Chopper collision, 93
Christmas tree, 25
Clearing the screen, 6
Collision detection, 92
Color, 6, 76
Color addressing, 18, 20
Color combinations, 8
Color graphics, 5
Color memory, 18
Computer talk, 146

Decimal frequency numbers, 117
Digital clock, 27
Disabling sprites, 67–69
Displaying a sprite, 51

Enabling sprites, 67–69
"Entertainer, The," 137

Filter mode, 133
Filters, 131
Flag, 96
Frequency settings, 102

Graphic characters, 3–4, 19–20
Gunfire, 148

Horizontal line, 21
Horizontal movement, 32, 38, 72

Independent duration, 125

Keyboard graphic, 8–9
Keyboard graphics, 3

"Largo," 114
Lunar lander, 23

Martians, 144
MSB register, 75
Multicolored sprites, 95
Multiple sprites, 66
Music, 101. *See also* Sound
Music octaves, 103

Normal mode, 4
Note duration, 106
Note values, 104

OR, 69

PEEK, 25
Pixel, 47
"Pizzicato Polka," 115
POKE, 17, 25, 27, 38
POKE codes, 19–20
Print strings, 6, 27, 31

Reaction timer, 10
Registers
  SID
    attack and decay, 108
    cutoff frequency, 133
    filter mode, 132
    frequency, 102
    octaves, 103
    pulse rate, 113
    resonance, 134
    sustain and release, 109
    voice to filter, 132
    volume, 102
    waveform, 111–13
  VIC-II
    background priority register, 90
    color, 76
    disable, 68
    enable, 67
    expansion, 77
    MSB register, 75
    multicolor register, 95
    sprite to background collision, 94
    sprite to sprite collision, 92
    X and Y coordinate, 71
Resonance, 134
Reverse graphics, 4, 20

Sailboat, 6–7
Saturn V rocket, 34
Screen addressing, 17, 20
Screen colors, 6
Screen coordinates, 71
Screen memory, 18
Select sprite program, 62, 82
SID, 102
Siren, 145
"Skater's Waltz," 126
Sound, 101
  arrays, 120
  attack and decay, 107
  decimal frequency numbers, 117
  filter mode, 133
  filters, 131
  frequency settings, 102
  independent duration, 125
  music octaves, 103
  note duration, 106
  note values, 104
  resonance, 134
  sustain and release, 109
  two- and three-voice music, 121
  voice to filter, 132
  volume settings, 102
  waveform settings, 111
Sound effects, 141
Sprite graphics, 47, 65, 85
  background to sprite priority, 87
  collision detection, 92
  disabling sprites, 67–69
  enabling sprites, 67–69
  horizontal movement, 72
  MSB register, 75

  multicolored sprites, 95
  multiple sprites, 66
  screen coordinates, 71
  sprite animation, 72
  sprite color, 76
  sprite data, 50
  sprite design, 48, 62
  sprite display, 51
  sprite movement, 71
  sprite numbers, 48
  sprite priorities, 85
  sprite registers, 65
  sprite size, 77
  sprite to background collision, 94
  sprite to sprite collision, 92
  vertical movement, 73
Steam engine, 147
Stickman, 41
Sunset, 90
"Surprise Symphony," 105, 107
Sustain and release, 109

"Take Me Out to the Ball Game," 110
Train, 35
Two- and three-voice music, 121
Typing graphics, 5

Vertical line, 21
Vertical movement, 33, 40, 73
VIC-II. *See* Sprite graphics
Voice to filter, 132
Volume settings, 102

Waveform settings, 111
White noise, 147
"William Tell Overture," 114

Sprite Design Grids

1								1								1							
2	6	3	1					2	6	3	1					2	6	3	1				
8	4	2	6	8	4	2	1	8	4	2	6	8	4	2	1	8	4	2	6	8	4	2	1

1								1								1							
2	6	3	1					2	6	3	1					2	6	3	1				
8	4	2	6	8	4	2	1	8	4	2	6	8	4	2	1	8	4	2	6	8	4	2	1

1								1								1							
2	6	3	1					2	6	3	1					2	6	3	1				
8	4	2	6	8	4	2	1	8	4	2	6	8	4	2	1	8	4	2	6	8	4	2	1

1								1								1							
2	6	3	1					2	6	3	1					2	6	3	1				
8	4	2	6	8	4	2	1	8	4	2	6	8	4	2	1	8	4	2	6	8	4	2	1

TO OUR READERS. . . .

Would you like to know more about microcomputer books published by Wm. C. Brown? If so, please provide the following information:

Name ____
Company (if applicable) ____________________________
Street ____
City ____________________ State __________ Zip ____________________
Most titles are available through better book and computer stores.

READER'S CRITIQUE. . . .

Wm. C. Brown wants to publish microcomputer books that meet your needs. Your feedback therefore is vital for us to improve current books and publish new and even better books in the future.

The feedback we need from you is outlined in the following questions. This page, when folded along the dotted lines in thirds, can be mailed back to us at no cost to you. Thank you in advance for the valuable information you are providing.

1. In which Wm.C. Brown microcomputer book did you find this form?
   Author: ____________________ Title: ____________________
2. Please rate this book in the following areas:

	Excellent	**Good**	**Fair**	**Poor**
a. Clarity of information	___	___	___	___
b. Completeness of explanations	___	___	___	___
c. Quantity of Examples	___	___	___	___
d. Usefulness of Examples	___	___	___	___
e. Degree to which Book Met Your Expectations	___	___	___	___
f. Overall Rating of Book	___	___	___	___

3. From what source did you purchase this book?
   ( ) Bookstore ( ) Convention Order ( ) Directly from Wm. C. Brown
   ( ) Computer Store ( ) Other: ______________
4. What brand(s) and model(s) of microcomputer(s) do you own/use?
   ____
5. What computer language(s) do you use? ____________________________
6. For how many years have you been using a computer?
   ( ) Less than 1 year ( ) 1-2 years ( ) 3-5 years ( ) More than 5 years
7. Which, if any, computer magazines/newspapers do you read regularly?
   ____
8. On what topics would you be most interested in purchasing microcomputer books? ______________
   ____
9. Please suggest ways in which WCB can improve future microcomputer books we publish.
   ____

QUESTIONS #10-14 ARE MERELY FOR CLASSIFICATION AND WILL BE USED ONLY TO DEVELOP A STATISTICAL PROFILE OF OUR RESPONDENTS.

10. Your occupation/job title? ____________________________
11. Do you use a computer in your work? ( ) Yes ( ) No
12. Your age?
    ( ) Under 18 ( ) 25-29 ( ) 40-49 ( ) 65 and over
    ( ) 18-24 ( ) 30-39 ( ) 50-64
13. Are you male or female? ( ) Male ( ) Female
14. What is the highest level of education you have attained?
    ( ) 8th grade or less ( ) High school graduate ( ) College graduate
    ( ) Attended/attending high school ( ) Attended/attending college ( ) Post graduate study

*****THANK YOU*****

wcb

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 551 DUBUQUE, IOWA 52001

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED
STATES

POSTAGE WILL BE PAID BY ADDRESSEE

WM. C. BROWN PUBLISHERS
Marketing Research Department
2460 Kerper Boulevard
P.O. Box 539
Dubuque, Iowa 52001

ABOUT THE BOOK

Create your own animation, play the piano, win at arcade style games, sing music, and much more. Now you can use all the sound, graphics, and music features of your Commodore 64. Become the expert you'd like to be in a step-by-step easy to understand approach. Loaded with illustrations and programs, this book can help you master sound, graphics, and music on your Commodore 64.

wcb

Wm. C. Brown Publishers
Dubuque, Iowa

$15.95 ISBN 0-697-00423-6